UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

PROYECTO FIN DE CARRERA

DESARROLLO DE UN VIDEOJUEGO DE PLATAFORMAS CON XNA Y PERSONAJES VIRTUALES BASADOS EN TÉCNICAS DE INTELIGENCIA ARTIFICIAL

Autor: *Jaime Chapinal Cervantes*

Tutor: *Jorge Muñoz Fuentes*

Septiembre 2011

El destino es el que baraja las cartas,
pero nosotros somos los que las jugamos.
- Arthur Schopenhauer -

Agradecimientos

En primer lugar, quiero agradecer a mis padres, Mari Carmen y José Luis, por su infinito apoyo y amor; por la educación que me han aportado y porque todo lo que tengo, y lo que soy, se lo debo a ellos.

A mi hermana, Libertad, por ser la mejor persona que existe en el mundo y por estar siempre dispuesta a ayudar.

A mis abuelas, Fabiana y Elisa, que en este duro año nos han dejado un hueco irremplazable en nuestras vidas. Por enseñarme tantísimos valores y ser un reflejo de vida. Estén donde estén, estoy totalmente seguro que estarán muy orgullosas de verme aquí.

A mis compañeros y amigos de la universidad, por haber hecho que este periodo de mi vida haya sido tan bonito. Y, en especial, a Javi, Carlos y Álvaro, por demostrar siempre que son unos excelentes amigos, estando tanto en los llantos como en las risas, que ha habido muchísimas a lo largo de estos años y espero que continúe siendo así siempre.

A mi tutor, Jorge Muñoz, por darme la oportunidad de realizar este proyecto donde he podido aprender muchísimo de unas de mis aficiones favoritas: la programación y los videojuegos. Por todo lo que me ha ayudado; y por estar siempre presente y con palabras de ánimo.

Y, por último, a todos los profesores y personas de las que he aprendido a lo largo de estos cuatro años.

Resumen

El proyecto trata sobre el desarrollo completo de un videojuego de plataformas en 2 dimensiones para PC a través del framework XNA. También se realiza un agente controlado por Inteligencia Artificial capaz de jugar al juego independientemente al que se enfrente. El algoritmo de búsqueda de caminos utilizado es el algoritmo A*. Dicho personaje estará diseñado para maximizar la puntuación, la cual se obtiene recogiendo monedas y llegando con rapidez a la meta. El jugador humano y el jugador del ordenador compiten en tiempo y lugar por conseguir la mayor puntuación.

Palabras clave:

Videojuegos ; Plataformas; Inteligencia Artificial ; Algoritmo A* ; XNA

Abstract

The project covers the complete development of a platform game in 2 dimensions for PC using the XNA framework. It also develops an agent controlled by artificial intelligence that can play the game regardless who is playing against it. The pathfinding algorithm used is algorithm A * or A star. This character is designed to maximize the score of the game, which is obtained by collecting coins, and to arrive to the goal as fast as possible. The human player and computer player compete in time and place to get the highest score.

Keywords:

Videogames; Platformers; Artificial Intelligence; Algorithm A*; XNA

Índice de contenidos

Índice de ilustraciones

Índice de tablas

Capítulo 1

Introducción

1. Introducción

El proyecto consiste en el desarrollo de un videojuego de plataformas y un agente que, mediante el uso de técnicas de inteligencia artificial, sea capaz de jugarlo. Ambos jugadores comparten tiempo y espacio en el nivel; y compiten por conseguir la máxima puntuación, la cual se obtiene recogiendo monedas y llegando lo antes posible a la meta.

Se pueden distinguir claras dos líneas de trabajo:

- Por un lado, el desarrollo completo del videojuego. Se empieza desde 0, con el objetivo de conocer y aprender todas las fases en la construcción de este tipo de software.

- Por otro lado, el desarrollo del agente de inteligencia artificial. Se ampliarán algunos de los conocimientos aprendidos en la asignatura de Inteligencia Artificial, con una aplicación directa en el problema. Se hace uso del algoritmo de búsqueda de caminos A*, con una heurística especialmente diseñada para este proyecto, la cual busca el equilibrio entre recoger monedas y llegar a tiempo a la meta.

Ambas líneas de trabajo convergen en el resultado final. Es decir, que se tendrá como resultado un producto jugable por usuarios, añadiéndole también se le añade la funcionalidad del agente con inteligencia artificial. Y es en este punto donde reside una de las novedades de este proyecto: el crear un videojuego de plataformas en dos dimensiones donde el jugador humano "compita", en el mismo tiempo y espacio, contra un jugador manejado por el ordenador. Dota de total originalidad al producto, ya que no se conoce ningún otro videojuego de plataformas con esta modalidad de juego en la que se compite contra un agente de IA. En todos los videojuegos de este género el usuario jugaba sólo o en modo cooperativo con varios jugadores; pero, con el único objetivo de completar los niveles

sin ninguna motivación extra. Con este juego, además de completar los niveles, se podrá competir contra el jugador de la máquina, para ver quien consigue mejor puntuación.

Se ha elegido también este proyecto por la aplicación real que se le puede dar a la inteligencia artificial en este tipo de videojuegos. Aunque bien es cierto que, la Inteligencia Artificial (IA) es factor clave en muchos géneros de videojuegos, como por ejemplo los shotter o de estrategia, nunca se le había dando un enfoque hacia los videojuegos de plataformas.

La idea del proyecto se vio muy influenciada también por la competición Mario AI Championship [1] . Es un campeonato que tiene como objetivo crear bots manejados por inteligencia artificial que sean capaz de jugar al juego Mario Inifinite (port Open Source del mítico juego Super Mario Bros), resultando ganador aquel que consiga máxima puntuación. El pasado año resultó ganador Robin Baumgarten con un bot que se puede observar en el siguiente vídeo [2]. En ciertas partes del proyecto se estudió su código para ver cómo lo hacía él en su bot, el cual se puede encontrar en la siguiente referencia [3] .

Ilustración 1 - Mario AI Robin

En un principio se pensó también en realizar un proyecto de este tipo: crear un agente que jugase al Mario Infinite y mejorar los resultados. Sin embargo, no se optó por esta opción por dos motivos:

1. Ya se había sobrepasado las fechas de inscripción a la competición Mario AI.
2. Además de la realización del bot, se quería desarrollar un videojuego completo.

Otro de los puntos desencadenantes de la realización de este proyecto fue leer la entrevista que hizo Robin Baumgarten en el portal especializado en Inteligencia Artificial en juegos, AiGameDev [4]. En dicha entrevista se puede leer lo siguiente:

> *Question: Can you envisage applying your solution to generate a behavior that maximizes points? Would there be any challenges there?*
>
> *Answer: This competition only rewards passing a level and does ignore any other metrics such as score and enemies killed…*
>
> *As a pure pathfinding problem, trying to get as many points as possible will be much more difficult to solve though, as it is not clear what the heuristic and immediate goals should be. However, I could imagine adding a planner on top of the pathfinding which guides Mario to each of the coins, boxes and enemies successively and lets him kill them. This would not be optimal at all, but because there is no time-penalty for taking longer (within the 180 seconds allocated for each level) it should work. It would be interesting to see if it's possible to optimise it such that Mario keeps jumping on Bullet Bills to get more points until the time almost runs out and then rushes towards the goal.*

Por tanto, se inspiró la idea de crear un algoritmo que fuese capaz de maximizar la puntuación. La diferencia con el proyecto de Robin, y a la vez lo que aporta una mayor riqueza computacional y originalidad, es que la heurística del juego se encargará también de recoger las monedas. El A* de la referencia anterior sólo se preocupaba de llegar lo antes posible a la meta. Sin embargo, en este proyecto se quiere abarcar esa idea también incluyendo la recolección de monedas, para obtener una mayor puntuación.

La ambientación y temática del videojuego se ha hecho sobre el personaje de ficción Enjuto Mojamuto [5]. Es un personaje que surgió a raíz del programa televisivo Muchachada Nui; y es bastante conocido en Internet, teniendo hasta una serie propia. Uno de los motivos por el que se decidió ambientar sobre este personaje es que, a fecha de comienzo del proyecto, no existía ningún juego sobre Enjuto. Posteriormente apareció el juego Enjuto Invaders en Facebook, un port del mítico "matamarcianos" Space Invaders. De cualquier modo, no existe ningún juego de plataformas sobre él; y el contexto del personaje está muy relacionado con la computación, Internet y los videojuegos.

Ilustración 2 - Enjuto Mojamuto

1.1 Motivación

La primera fase del proyecto fue la determinación de objetivos e ir acotando el problema en función de éstos. La idea de realizar un videojuego y agentes de Inteligencia Artificial que lo jugasen estaba clara, pero había que decidir sobre cuál hacerlo. Se barajaron distintas posibilidades:

- Super Mario Bros – Clon del juego realizado en Java. Se basaba en realizar un agente para que jugase los niveles aleatoriamente generados para la competición Mario AI [1]

- Torcs – Videojuego de coches. Consistía en la realización de un bot que seleccionase las mejores características del coche en función de varios parámetros.
- Ms. Pac-Man – Agente que por IA jugase a este mítico juego.
- Starcraft – Agente IA que jugase a este juego de estrategia.

Aunque no se optó por ninguna de las anteriores, se inspiró en la idea de SuperMarioBros para que el juego fuese un plataformas en 2 dimensiones.

Mis motivaciones personales para la realización del Proyecto Fin de Carrera sobre videojuegos es que, desde siempre, estos han sido mi pasión. Desde muy niño pasé grandes ratos jugando; y, a medida que iba creciendo y me iba acercando más al mundo de la informática, mi interés fue desviándose hacia su desarrollo. Cada vez que descubría un juego nuevo, me asaltaba la misma pregunta ¿cómo lo habrán hecho? Puedo decir que los videojuegos, junto a Internet, fueron los principales motivos por los que decidí estudiar Ingeniería Técnica Informática.

Ya en la carrera, las asignaturas que siempre me llamaron más la atención y más me gustaron fueron las de programación. Hasta entonces, nunca había programado, y la verdad es que me gustó tanto que de aquí en el futuro me gustaría orientar mi carrera profesional hacia esta rama. En último curso tuve la asignatura de Inteligencia Artificial; y también puedo decir que, aunque llevó mucho trabajo y esfuerzo, la considero una de las materias más interesantes de las que he aprendido en estos años. Y, sobre todo, porque veía clara y directamente su relación con el mundo de los videojuegos, algo que me llamaba desde hace tanto tiempo. Por ejemplo, las prácticas consistían en una adaptación del famoso videojuego de PC "Los Sims". Y, aunque tenía detrás una gran carga teórica, la verdad es que disfruté trabajando en ello.

Por todos estos motivos, decidí orientar mi PFC hacia temas que me resultan tan interesantes y atractivos; además de que cuentan con un gran futuro en el mercado actual como se plasmado en la introducción de esta sección.

1.2. Objetivos

El objetivo principal de este proyecto es conocer y aprender todas las fases del desarrollo de un videojuego de plataformas en 2 dimensiones. Todo ello, a través de la tecnología de desarrollo XNA de Microsoft, ampliamente utilizada para el desarrollo de juegos (sobre todo a nivel de desarrolladores independientes). Además, XNA tiene una amplia comunidad de desarrolladores en Internet, denominada XNA Creators, en la cual se ofrecen diferentes recursos de apoyo así como soporte entre usuarios.

Además del desarrollo íntegro del juego, se tiene también el objetivo de crear un agente que por técnicas de Inteligencia Artificial sea capaz de jugar al videojuego en cuestión, sea cual sea el nivel al que se enfrente. El algoritmo con tal fin es el algoritmo A*, ampliamente utilizado en otras aplicaciones de la IA, como por ejemplo la búsqueda de caminos entre dos puntos en los Sistemas de Posicionamiento Global (GPS). De esta forma, se persigue profundizar en los conocimientos sobre Inteligencia Artificial y como, esos conceptos teóricos, pueden ser aplicados a la práctica por medio de videojuegos.

Por tanto, el alcance del proyecto se clasifica, en dos bloques fundamentales:

1. Completo desarrollo de un videojuego de plataformas.
2. Diseño e implementación de un bot capaz de jugar al videojuego mediante usos de técnicas de inteligencia artificial.

En base a este objetivo principal, se proponen los siguientes objetivos parciales:

- El juego tiene que ser completamente funcional; ha de poderse jugar en un ciclo completo sin errores que hagan fallar a la aplicación.

- Jugabilidad. Se tiene que poder jugar y que "aporte diversión" al usuario.

- El código del juego tiene que ser escrito para facilitar una posible continuación futura con nuevas funcionalidades. Todos los métodos y propiedades relevantes han de estar comentadas.

- La IA desarrollada tiene que ser eficiente y estar optimizada. De nada sirve tener un algoritmo perfecto, si no es capaz de ser ejecutado en tiempo real; ya que haría el juego totalmente inestable.

- Se facilitará, en la medida de lo posible, una futura portabilidad a la plataforma Xbox 360.

- Se contará con un módulo de puntuaciones para los jugadores, donde podrán consultar las máximas puntuaciones de cada nivel y entrar en el ránking si logran estar entre los 5 primeros.

- Para el análisis y estudio de la IA, se proporcionarán métodos de log; donde queden reflejados todos los resultados relevantes del algoritmo A* mediante trazas.

- El usuario podrá añadir fácilmente niveles, con el objetivo de que pueda crear él mismo sus propios niveles como quiera y no se limite a los que vengan pre-configurados en la instalación.

1.2. Estructura del documento

El presente documento se organiza siguiendo la siguiente estructura:

- Capítulo 1: *Introducción* – Se explica el problema que va a abarcar el proyecto, los objetivos los cuales se quieren conseguir y las motivaciones que han llevado a la elaboración del mismo.

- Capítulo 2: *Estado del arte* – Realiza un breve estudio general sobre la industria de los videojuegos; para a continuación centrarse en los juegos de las plataformas, con un recorrido sobre la historia de este género. Además, se muestra una estadística con el volumen de ventas de los títulos más importantes. El capitulo finaliza con una visión general de la Inteligencia Artificial aplicada a los videojuegos, ofreciendo algunos ejemplos sobre IA en otro géneros, como por ejemplo el fútbol. Se estudian también las repercusiones que tiene la IA sobre los videojuegos en su globalidad; acabando con técnicas IA específicas que son usadas en la elaboración de videojuegos.

- Capítulo 3: *Descripción* del juego – En este capítulo se expone una descripción textual del videojuego en cuanto a cómo va a ser y cuál va a ser su mecanismo de funcionamiento. A continuación se prosigue con la herramienta elegida para su desarrollo, XNA, argumentando por qué se ha elegido y cuál son sus características principales.

- Capítulo 4: *Desarrollo* – En esta sección se muestran todos los apartados técnicos que han abarcado las tareas de desarrollo del proyecto: desde su análisis (incluyendo casos de usos y requisitos), pasando por el diseño en la arquitectura del sistema y modular, y acabando en la implementación a más bajo nivel. Por último, se muestran las pruebas realizadas sobre el estudio de la eficiencia del algoritmo mediante sus trazas; y los test de prueba a los que se vio sometido el juego para verificar su correcto funcionamiento.

- Capítulo 5**:** *Conclusiones* – En este capítulo se muestran todas las conclusiones finales que se sacan a raíz de la elaboración del proyecto.

- Capítulo 6: *Trabajos futuros* – Abarca una serie de ideas que se proponen para la continuación del proyecto con nuevas funcionalidades.

- Apéndices**:** Se muestran los apartados referidos a la *planificación* y *presupuestado* del proyecto, tanto el inicial como el real, y se hace una comparativa de la desviación entre ambos. En esta sección también se incluye el *manual de usuario* y los medios que han sido empleados en el desarrollo.

- Bibliografía y referencias: Última sección del documento donde se recoge y agrupa todo el material consultado para este proyecto.

Capítulo 2

Estado del arte

2. Estado del arte

2.1 Panorámica del mundo de los videojuegos

El mundo de los videojuegos ocupa una posición primordial y privilegiada en el sector del entretenimiento. Además, en nuestros días, y en la sociedad en la que vivimos, el ocio cada vez va ocupando un espacio mayor en nuestras vidas. Este hecho ha venido motivado también por la aparición, sobre todo a lo largo de este último siglo, de nuevas formas de entretenimiento, como por ejemplo podrían ser la fotografía, el cine, o la televisión. Desde la aparición de los videojuegos, hasta ahora, dicho sector no ha hecho más que seguir su línea ascendente. Y, es más, en la época actual de profunda crisis mundial en la que nos encontramos, es de los pocos sectores que se mantiene estable, e incluso progresando.

Como muestra de lo expuesto, se cita el estudio que ha realizado la empresa consultora de y de investigación de mercados tecnológicos Gartner Inc en el artículo de la referencia [6] :

> *"en 2011 se gastarán 74.000 millones de dólares (51.137 millones de euros) en esta forma de entretenimiento. Esta cifra supone un incremento del 10,4% con respecto al gasto en el pasado año 2010, el cual ascendió hasta los 67.000 millones de dólares. Además, se pronostica que para el 2015 la industria generará 115 mil millones de dólares anuales."*

Otra referencia a ésta misma información que apareció en Informativos Telecinco [7]

El centro investigador de la Universidad de Palermo (Argentina) ha publicado también un interesante artículo [8] en el que se recogen, entre otras, estas conjeturas del crecimiento de la industria del videojuego. Califica al sector como muy dinámico y vivo.

Estadísticas muy significativas se pueden obtener de la organización ESRB (Entertainment Software Rating Board), [9] la cual se encarga de determinar la edad apropiada para los juegos que se publican. Han hecho un estudio sobre los hábitos del uso de videojuegos en EEUU en 2010. Destacan, entre otros, los siguientes datos, los cuales pueden ser consultados en la referencia: [10]

- El 67% de los hogares de EE.UU. juega a videojuegos.
- El rango de edad medio es de 18-49 años.
- Se juega de media 8 horas por semana.
- 40% de los jugadores son sexo femenino, y la gran mayoría de éstas utilizan la videoconsola Nintendo Wii.
- Los videojuegos son el entretenimiento en el que los padres están más pendientes de sus hijos, más que en cine, Internet y televisión.
- Los videojuegos de consola generan muchísimos más beneficios que los videojuegos de ordenador (9.9 billones de dólares frente a medio millón.

Y todo esto no es, para nada, sólo en Estados Unidos. Este hecho se extrapola a nivel mundial, teniendo como referencias en el mercado oriental a Japón; y en el europeo al Reino Unido. Y, en España, ocurre igual; aunque, como en muchos otros aspectos del país, a un ritmo menor que del resto de potencias. Sin embargo, a relación es interesante destacar iniciativas que se empiezan a realizar en nuestro país con el fin de dar más vida al mundo del desarrollo de videojuegos, tal y como se puede apreciar en los siguientes artículos [11] [12]

Nos encontramos, pues, en un buenísimo momento para la industria del videojuego, tal y como señala el presidente de la compañía Electronic Arts en [13]

Ilustración 3 - Videojuegos y economía

Sin embargo, este sector va evolucionando, y no sólo por la "gran" industria. Con el crecimiento de la tecnología, cada vez va tomando más fuerza los desarrollos de los llamados" juegos indie" (independientes) [14]. Cada vez son más frecuentes estos tipos de juegos desarrollados por personas individuales, o pequeños equipos. Son una grandísima oportunidad para toda la gente que no está dentro en las grandes compañías de videojuegos, y quieren desarrollar su propio producto. En cierto modo, se podría decir que es una "democratización" de los videojuegos.

Esta situación ha aportado muchísimos aspectos positivos. Al haber cada vez más gente disponible para realizar sus obras, hay mucha más oferta y variedad en el mercado. Esto genera, inevitablemente, que la competencia sea mucho más difícil; y, a la vez, que factores "puros" de los videojuegos, como la diversión o el entretenimiento, unidos a la originalidad, sean los pilares fundamentales del éxito del producto.

En ciertos aspectos, como por ejemplo sería el multimedia, es muy difícil que un juego indie supere a uno desarrollado por una gran empresa del sector. Esto es una consecuencia directa de los recursos de los que disponen ambos. Sin embargo, es muy interesante observar cómo lo que en un principio fueron "pequeños" juegos indie, crecen y se convierten en referencias del momento. Un claro ejemplo de esto serían los juegos

Minecraft [15] y Plantas vs. Zombies [16]. Este último es un ejemplo significativo de cómo una pequeña compañía indie (PopCap Games) puede ser absorbida, debido a su éxito, por otras gigantes compañías, en este caso Electronic Arts por la cifra de 750 millones de dólares. [17]

Ilustración 4 - Plantas vs Zombies

Las grandes compañías se han percatado de este hecho, y han apostado por él. Ofrecen SDK's (Kits de desarrollo software) propios para que los usuarios programen para sus plataformas. Además, ofrecen también la posibilidad de comercializar a través de sus herramientas los juegos. Claros ejemplos de esto son:

- Microsoft – Xbox 360 - Xbox indie Games [18] Desarrollo con XNA.
- Nintendo – Wii Ware [19]
- Apple - SDK de pago (100$ anuales) y posibilidad de venta en la AppStore. [20]
- Android Market [21]

2.2 Videojuegos de plataformas

Definir el concepto o sub-género de *videojuego de plataformas* puede ser una difícil tarea por la evolución que éstos han sufrido y cómo han ido desembocando y uniéndose con otros muchos subgéneros, como acción, aventuras, RPG o puzles. Sin embargo, una definición perfectamente

correcta que engloba fielmente al concepto en general es el que dice la web sobre videojuegos elotrolado.net (uno de los portales sobre videojuegos en habla hispana más visitados del mundo) en [22]:

> *"Género de videojuegos en el que el personaje controlado por el jugador debe saltar, subir o bajar por diferentes "plataformas", elevaciones o depresiones del terreno de amplitud variable. Estas plataformas pueden ser estáticas o estar en movimiento."*

Por tanto, los videojuegos de plataformas se caracterizan por el control de un personaje a través de los niveles, con el objetivo principal de alcanzar el final para poder superar la partida. Los tipos de movimientos que el personaje puede realizar depende del videojuego en cuestión; sin embargo, la acción elemental e imprescindible es la de salto.

A lo largo del nivel, existen diversos elementos que dotan al juego de mayor diversión. Cada videojuego tiene sus propias características; sin embargo, éstas se podrían clasificar en dos bloques principales:

- Enemigos – Su objetivo es enfrentarse al personaje del jugador para matarle e impedirle que llegue a final del nivel. En cada videojuego suele haber distintos tipos de enemigos, cada uno con sus características particulares. Del mismo modo, depende del videojuego, el personaje principal puede matar a sus enemigos; o, simplemente, evitarlos. Los enemigos no suelen tener mucha inteligencia; y su aumento de dificultad a medida que se va avanzando en los niveles suele radicar en que es más difícil de evitarles o matarles. En algunos juegos, también se incluye un enemigo al final del nivel que es conocido como "jefe final" y es más complicado que los enemigos "normales".

- Ítems – A lo largo del nivel, el personaje va recogiendo estos ítems. A su vez, éstos se podrían clasificar también en dos categorías en función de su utilidad. Por ejemplo, existen

power ups y power down. Los primeros dotan al personaje de características especiales que ayudan al personaje a llegar al final con determinadas facilidades, según el power-up. Ejemplo: inmunidad, mayor rapidez de desplazamiento y salto. Los Power-down causan el efecto contrario, es decir, impedir que llegue a la meta de manera normal, como por ejemplo decelerar la velocidad de movimiento o saltos menos potentes.

- Puntuación – Proporciona puntos al jugador, los cuales se registran en el contador de puntos del jugador.

- Tiempo – Los niveles disponen de un tiempo máximo para ser superados; sino, la partida finaliza.

2.3 Historia de los videojuegos de plataformas

La base de la información ha sido sacada del siguiente artículo [23]. Existe cierta controversia en el mundo de los videojuegos sobre cuál puede considerarse como el primer videojuego de la historia. La "discusión" se centra entre los videojuegos Space Panic, y Donkey Kong.

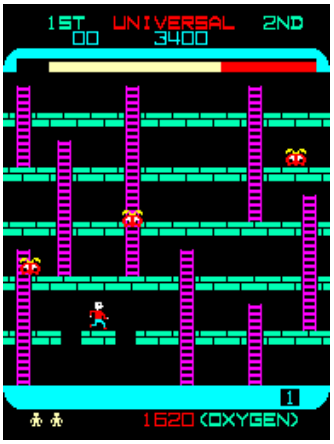


Ilustración 5 - Space Panic

Space Panic [24], desarrollado por la compañía nipona Universal en el año 1980, es un videojuego arcade [25] en el que el personaje de un astronauta debe moverse por el nivel para capturar a los alienígenas. Aunque el personaje no puede saltar, hecho por el cual se le cuestiona como primer plataformas de la historia, sí existen plataformas en distintos niveles por las cuales el personaje accede a través de escaleras y cavando hoyos. También introduce el factor de tiempo de nivel, en forma de oxígeno que le queda al astronauta para superar el nivel.

Al año siguiente, en 1981 también empresa japonesa Nintendo [26] sacó a la luz el arcade Donkey Kong [27]. Por primera vez se introducen los saltos a través de las plataformas. Es en esta diferencia entre ambos juegos, donde reside el debate sobre cuál fue el primer videojuego de plataformas. Dada la actual visión y definición de este género, la cual está basada en el salto entre distintas superficies y evitar obstáculos como principal movimiento, se suele considerar a Donkey Kong como el precursor de este tipo de videojuegos. No obstante, es innegable y obvia la clara influencia que tuvo el videojuego Space Panic sobre él.

Ilustración 6 - Mario Bros.

El nombre del juego venía del malvado personaje, Donkey Kong, el cual había secuestrado a la princesa Pauline. Un personaje llamado Jumpman tenía el objetivo de rescatarla. Éste es el origen del *archiconocido* personaje Mario, del cual se hablará próximamente. El jugador controlaba a

Jumpman, y debía saltar de plataforma en plataforma, utilizar escaleras y ascensores; además de sortear los obstáculos, por ejemplo en forma de barril, que le lanzaba Donkey para alcanzar a la princesa. Por el camino, podía recoger ítems los cuales sumaban puntos. También se podían conseguir puntos por superar obstáculos, recoger power-ups y otras tareas. El juego tenía 4 pantallas a superar para poder completar el juego y rescatar a la princesa. Fue un absoluto éxito y llegó a vender 65.000 copias en Estados Unidos [28].

Ilustración 7 - Donkey Kong

Al año siguiente, 1982, se lanzó la secuela del videojuego, llamada Donkey Kong Jr. [29]. En esta ocasión, cambian los roles respecto a la anterior entrega: Donkey Kong ha sido capturado por Mario. Esta es la primera ocasión donde aparece el nombre de Mario, en lugar del anterior Jumpman. Es un hecho curioso el que el personaje principal de las sagas Super Mario Bros (las cuales comentaré más adelante), tenga su primera aparición en otro videojuego distinto, como sucede en este caso. El jugador controla a un nuevo personaje, Donkey Kong Jr., hijo de Donkey Kong, con el objetivo es liberar a su padre. El modo de juego sigue las directrices marcadas por la edición anterior del juego, Donkey Kong, comentada en el párrafo anterior. Como novedad, y a relación con el argumento del juego, se introduce que Donkey Jr. puede trepar a través de lianas.

Ilustración 8 - Cartuchos Donkey Kong Famicom

Estos juegos ya comentados fueron sacados para máquinas arcade. Sin embargo, a mediados del año posterior (15 Julio de 1983) se encuentra un hito en la historia de los videojuegos transcendental: la aparición en el mercado nipón de la videoconsola Famicom [30]. Y destaco este hecho, además, porque su lanzamiento fue junto a los videojuegos de plataformas que han sido estudiados en este documento, *Donkey Kong, Donkey Kong Jr.*, junto a otro título, *Popeye* [31]. La consola pasó a denominarse NES [32] (Nintendo Entertainment System) al desembarcar en el mercado norteamericano (1985), europeo (1986) y australiano (1987).

Ilustración 9 - Consola NES

Todos los juegos comentados hasta el momento, tenían una característica común: la cámara. Utilizaban una cámara fija, es decir, que siempre se mostraba por pantalla lo mismo; y era el nivel completo. El primer videojuego en cambiar este hecho, y aportar mayor jugabilidad y diversión fue Jump Bug [33]. Lo cual permitía tener un nivel mucho mayor moviendo la cámara en función al personaje del jugador. Se podría decir que hasta la aparición de este videojuego, no terminó de definirse el concepto de *videojuego de plataformas*, ya que el scroll vertical tuvo una importancia más que fundamental en los juegos venideros. Fue lanzado en el 1981, 5 meses después de Donkey Kong. El jugador controlaba un coche, el cual podía disparar a los enemigos o saltar para evitarlos. Introdujo el scrolling horizontal, side-scrolling [34], ya que el coche iba avanzando hacia la derecha hasta completar el nivel. La cámara iba siguiendo y

manteniendo centrado siempre al coche. Éste, además, siempre iba saltando, pero se podía controlar el salto con las teclas de arriba y abajo.

Ilustración 10 - Jump Bug

Posteriormente, en 1982, salió al mercado el título Pitfall! [35]. Éste introdujo el *scrolling horizontal;* pero en un formato diferente, ya que las pantallas del nivel se iban sucediendo. Al llegar al final de la pantalla, se cargaba una nueva, la cual era continuación de la anterior y del mismo nivel. Esto implicaba que no tenía una cámara que seguía al personaje, como era el caso de Jump Bug. Este tipo de scroll daría años siguientes pie a la aparición de ostros juegos que utilizaban este sistema. El máximo referente de esto es Prince of Persia. Fue desarrollado por Activision, lanzado y un verdadero éxito de ventas. El mismo año saldría el título Jungle King, que utilizaba este sistema también pero con algunas mejoras. Por ejemplo, en las fases de saltar por las lianas, utilizaba el sistema de scrolling *exploración* de Pitfall! pero con un movimiento de cámara mucho más suave. No ponía la pantalla a negro y volvía a cargar la escena, como hacía Pitfall, sino que al llegar al final de la escena, mantenía siempre en escena al personaje mientras realizaba el cambio de cámara. En el siguiente enlace se puede observar un vídeo [36] de lo explicado.

Ilustración 11 - Pitfall!

Otros de los primeros juegos de plataformas, que también añadieron la novedad de múltiples pantallas por nivel, fueron Smurf: Rescue in Gargamel's Castle [37] y Jet Set Willy [38].

A finales de ese mismo año, aparecía Moon Patrol [39], otro juego de plataformas similar a Jump Bug. La novedad que introdujo este título fue que se convirtió en el primero en implantar el parallax-scrolling [40] en el fondo del nivel [41].

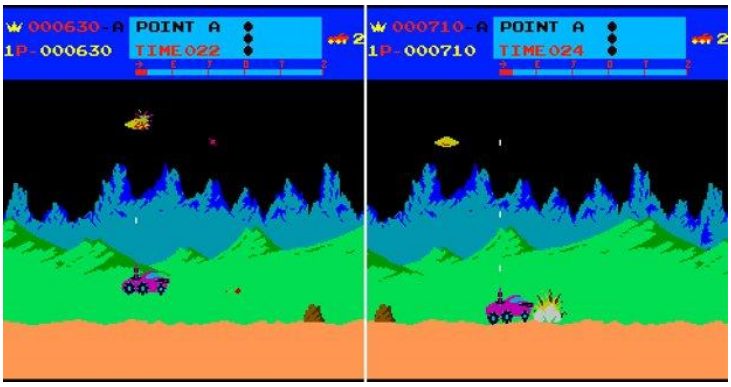


Ilustración 12 - Moon Patrol

En 1983 surge también la primera entrega de, lo que se convertiría en el precursor de una de las mayores sagas de videojuegos, Mario Bros [12] . Lo desarrolló Nintendo en un principio como juego arcade, aunque pronto tendría su conversión a la plataforma NES. El personaje que el jugador manejaba era el fontanero Mario [13], del cual hablamos anteriormente que había tenido su primera aparición en los juegos de Donkey Kong. Sin embargo, introduce una importante novedad y era el multijugador cooperativo en los juegos de plataforma. Podían jugar dos jugadores, utilizando éste último al compañero fontanero de Mario, llamado Luigi.

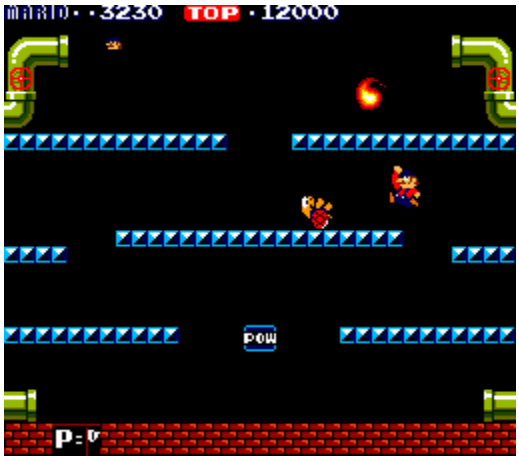


Ilustración 13 - Mario Bros.

Fue, por tanto, el pionero en establecer este modo de juego con varios jugadores que tanto desarrollo y posibilidades daría en el futuro, por ejemplo con Bubble Bobble en el año 86. Éste juego estaba claramente influenciado por Mario Bros., ya que tenía también el modo cooperativo y era en pantalla fija. El juego consistía en recuperar a las novias de Bub y Bob (los protagonistas). Para ello, se ayudaban con el lanzamiento de burbujas para capturar enemigos y completar el nivel. Una característica muy importante también de este juego es que fue el primero en incluir múltiples finales, dependiendo de cómo se hubiese desarrollado la partida. Por ejemplo, el final dependía de si habían sobrevivido dos jugadores, o sólo uno; y si habían completado las fases de bonus o no. Es otro gran clásico de los videojuegos y ha sido adaptado a distintas plataformas en el futuro.

Ilustración 14 - Bubble Bobble

En el año 1983 salió el videojuego B.C.'s Quest For Tires, el cual se caracteriza porque empieza a haber una progresión del género hacia la velocidad a lo largo del nivel y, sobre todo, la precisión del salto.

Al año posterior, 1984, aparecieron dos títulos muy importantes. El primero de ellos fue Pac-Land. Es una evolución en el género e influyó mucho en los juegos venideros. Además, combinaba y optimizaba distintos elementos que estaban viendo en juegos anteriores, aportando su granito de arena para mejorarlos, por ejemplo en el parallax-scrolling y en el scrolling. También es de ese año Legend of Kage, que ofrecía la novedad de juegos que se podían desarrollar en cualquier dirección.

Pero, uno de los grandes hitos de la historia de los videojuegos de plataformas vino en 1985. Es un hito muy importante porque en la fecha del 18 de Septiembre de 1985 sale al mercado el videojuego que marcaría una generación, Super Mario Bros. [42] Muchísimo se podría hablar de este videojuego que tan crucial ha sido en este mundo; pero, se mostrarán algunas pinceladas de lo que ha sido el juego: Es el juego con mayor ventas de la historia; en 1999 entró en el Libro Guiness de los Récords por vender más de 40 millones de copias. Se considera tan importante y significativo este juego que, aún hoy en días, es común que en la jerga se nombre a los videojuegos de plataformas como los "tipo Super Mario". Se maneja al fontanero Mario para rescatar a la princesa Peach de las garras del monstruo Koppa. También se puede jugar dos jugadores con Luigi, aunque por turnos. El juego consta de 8 mundos y su primera aparición fue para la consola NES. Y, otro aspecto que no se puede obviar, es que sirvió como base para una de las sagas más famosas, por no decir la más, de la historia de los videojuegos. A partir de éste, surgieron otros muchos juegos de la misma familia que siguieron con el éxito del primero.

Ilustración 15 - Super Mario Bros.

Tras este impresionante éxito, la compañía Sega respondió con dos juegos también importantes (además del ya comentado Bubble Bobble de Taito). El primero de ellos, *Alex Kidd*, [43] se caracterizaba por largos niveles a explorar pero no sólo en horizontal, sino que también en vertical. Además, podía romper bloques y rocas del nivel para abrirse paso. Y, otro hecho que lo destaca, es que fue de los primeros en los que se podía seleccionar distintos niveles por medios de password. Esto fue un avance, ya que hasta la fecha no había modo de "guardar" el juego. Por ejemplo en Mario, si querías completar el juego entero, lo tenías que hacer de seguido, con todo el tiempo que ello requería y exponiéndote a posibles incidencias como el fallo de la luz. Sin embargo, de esta forma, cada vez que completabas un nivel se te facilitaba un Password para comenzar desde el siguiente nivel con dicha clave. Esto se portó a muchos juegos a partir de entonces.

Ilustración 16 - Alex Kidd

El otro título importante del 86 de mano de Sega fue Wonder Boy [44] Al tener mucho éxito, también gozó de múltiples secuelas y actualizaciones. Estaba muy inspirado en Pac-Land y utilizaba elementos de patinaje para aportar mayor sensación de velocidad. Esto hacía que los niveles aumentasen su dificultad.

Ilustración 17 - Wonder Boy

Este año es importante también por la aparición del juego de Nintendo Metroid [45]. La gran característica de éste es la combinación de los juegos de plataformas con la libre exploración del mapa.

Ilustración 18 - Metroid

1987 se caracteriza por la aparición de otro de los juegos de plataformas por excelencia, Megaman [46] desarrollado por Capcom. Una

de las novedades que introducía este juego en escena era la libre selección de niveles, evitando así los juegos lineales como Super Mario Bros.

Ilustración 19 - Megaman

El último año de los 80, viene marcado por la aparición del clásico plataformas Prince of Persia. Aportaba un gran salto de calidad en el juego, y con una historia fascinante. Fue portado a muchísimas plataformas; y combinaba aspectos de otros géneros como podrían ser los puzles.

Ilustración 20 - Prince of Persia

El nacimiento de las consolas de 16-bit tuvo su repercusión directa en el género de videojuegos que nos ocupa. Nintendo sacó su plataforma Super Nintendo (SNES) [47] el 21 de Noviembre del año 1990 en Japón. Su lanzamiento coincidió con el videojuego Super Mario World [48]. Los mayores aspectos a destacar de este título son la jugabilidad y la diversión. Seguía siendo el prototipo de juego de plataformas, pero aumentaba muchísimo el número de niveles por mundo y los ítems y power-ups a usar por Mario, al igual que el número de enemigos.

Ilustración 21 - Super Mario World

Por su parte, Sega había sacado su consola de 16 bit hacía 2 años, en 1988. Sin embargo, su mayor despegue se produjo también en el 90 por la aparición del juego Sonic the Hedgehog [49] protagonizado por el erizo Sonic, el cual se convertiría en personaje *abanderado* de la marca Sega. El principal factor que lo definía era la velocidad. Sonic podía rodar y conseguir altísimas velocidades, lo cual dotaba al juego de un alto dinamismo y requería bastante agilidad visual para esquivar enemigos y obstáculos en esas condiciones. También introdujo una característica que se tomará, en parte, en la realización de este proyecto, y es el modo de puntuar y las vidas. Otro aspecto novedoso en la época fueron los mapas y su buen motor físico que los soportase. Un claro ejemplo de esto eran los loopings que Sonic debía tomar a gran velocidad para poder superarlos.

Ilustración 22 - Sonic the Hedgehog (looping)

Se iban recogiendo anillos, y cuando un enemigo tocaba a Sonic éste perdía los anillos los cuales se repartían por las cercanías durante un tiempo

determinado. Si no se recogieran, desaparecían. En el caso que Sonic fuese golpeado cuando no disponía de ningún anillo, se acababa la vida del personaje y había que volver a empezar desde el principio o desde un punto de guardado por partida. En este modo se ha inspirado el proyecto para tratar también las colisiones con enemigos. Fue de los primeros grandes éxitos de Sega Megadrive; y tras estos acontecimientos, se decidió sacar una versión en 8 bits para la consola Sega Master-System.

Ilustración 23 - Sonic the Hedgehog

Por otro lado, para el sistema operativo MS-DOS, apareció el videojuego Commander Keen. [50]

Con el establecimiento de consolas como PlayStation, Nintendo 64 o Sega Saturn, empezaron a usarse más juegos en tres dimensiones, en detrimento de los juegos de 2D. Sin embargo, se intentó combinar ambas tecnologías, dando lugar a la denominada 2.5D, en la que se combinan elementos en 3D con otros en dos, por ejemplo entre fondos y personajes. El primer juego en llevarlo a cabo fue Clockwork Knight, de Sega Saturn. Sirvió de influencia a muchos juegos posteriores como Klonoa o Pandemonum.

Ilustración 24 - Clockwork Knight

El primer videojuego "puro" de plataformas en 3D fue Alpha Waves [51] (también llamado Continuum), el cual salió en 1990. Era un juego que combinaba las plataformas con los laberintos. Aunque tenía un aspecto simple, fue el primero en que trabajaba solamente con gráficos en tres dimensiones, así como su movimiento de cámara. Fue, por tanto, el primero en intentar adaptar el género completamente a la tecnología 3D.

En el año 1995, sale al mercado para la consola PlayStation el juego Jumping Flash! [52] Era un juego de plataformas desarrollado también totalmente en 3D, y con unos gráficos mejorados para la época y con gran jugabilidad. Ese mismo año aparece también el juego Bugs! [53] para Sega Saturn, con características similares. Estos dos títulos son, para muchos, los primeros videojuegos de plataformas en 3 dimensiones, porque consideran que Alpha Waves le faltaba para ser considerado un juego de plataformas.

Al año siguiente, 1996, PlayStation saca a la luz el juego protagonizado por el personaje que se convertiría en uno de las insignias de la consola, Crash Bandicoot [54]. Fue un total éxito, ya que ofrecía muchísima jugabilidad a lo largo de sus niveles. Seguía manteniendo la linealidad de los juegos de plataformas de antaño, pero con unos gráficos mucho más modernos y en tres dimensiones. Ofrecía toda la esencia de los juegos de plataformas; añadiendo, además, nuevas funcionalidades al género que lo hacían aún más interesante. La novedad es que posicionaba la cámara detrás del jugador, y así se podía seguir su perspectiva. Otra novedad fueron, por ejemplo, los niveles en los de persecución, en el que

Crash tenía que huir de enemigos como osos, u elementos como rocas. En estos casos, la cámara se posicionaba delante del jugador y éste tenía que correr hacia ella, consiguiendo un efecto de velocidad hacia el usuario que está jugando, como muestra la siguiente imagen:

Ilustración 25 - Crash Bandicoot

También en ese año, ocurre un hito importante con Nintendo, ya que saca a la venta la consola Nintendo 64, con más potencia de procesamiento que las existentes hasta la fecha. Junto a ella, se produce el lanzamiento del videojuego Super Mario 64, la cual es la primera adaptación del mítico fontanero a las 3D. El juego era completamente tridimensional, y aportaba como novedad un sistema de cámaras, muy útil para el usuario, que se controlaba con el stick analógico. Con este juego se introdujo también otro tipo de objetivos para los juegos de plataforma no visto hasta entonces: la recolección de ítems. Hasta entonces, el único objetivo que se tenía en los juegos de plataformas, era llegar hasta el final (y consiguiendo puntos también). Sin embargo, con la libertad que proporcionaba el movimiento en 3D en SuperMario64, se introduce el objetivo de buscar y conseguir estrellas, para pasarse determinadas fases del juego. Este título fue una gran influencia para otro gran éxito de esta consola, el Banjo Kazooie.

Ilustración 26 - Super Mario 64

El otro gran icono de los juegos de plataformas, Sonic, no hizo el salto al 3D hasta finales de1998, cuando apareció el juego Sonic Adventure [55] para la consola Dreamcast. Seguía el nuevo estilo de juego que había comenzado Super Mario 64. Sin embargo, la novedad que este juego ofrecía era, como venía siendo habitual en la saga Sonic, la sensación de velocidad. Ésta, unida a los gráficos 3D daba unos resultados muy interesantes.

Ilustración 27 - Sonic Adventure

En esta época caben destacar, también, los siguientes títulos de plataformas que aparecieron para las consolas de Sony PlayStation y PlayStation 2: Spyro the Dragon (1998), Rayman (1995), Jack & Dexter (2001) y Ratchet & Clank (2002). Por su parte, Nintendo, con su nueva consola Game Cube, lanzó la nueva secuela de Mario, Super Mario Sunshine.

Con la última generación de consolas (PlayStation 3, Xbox 360 y Wii) los juegos de plataforma también han tenido su hueco; aunque, bien es verdad, que han sido en parte desplazados por juegos que buscan aprovechar todo el potencial de las consolas en el apartado gráfico.

Para PlayStation 3, aparte de las continuaciones de juegos como Ratchet & Clank o Jak & Dexter, el máximo exponente del género es, sin duda, Little Big Planet. Este juego aporta un factor novedosísimo: el usuario puede crear sus propios niveles. Esto, unido a la conexión a Internet de la consola, permite también que los jugadores de todo el mundo compartan sus niveles. Este juego también salió para la consola portátil de Sony, la PSP.

Ilustración 28 - Little Big Planet

Por su parte, Nintendo, ha continuado con su tradición y ha sido la que más partido ha sacado a sus juegos de plataformas. Destacan, por encima de todos, las continuaciones de sus grandes clásicos, como por ejemplo, Super Mario Galaxy. Daba una vuelta de tuerca más a la jugabilidad de plataformas en 3 dimensiones. Esto es debido a la novedad que se le añadió: distintas gravedades.

2.3 Estadísticas videojuego de plataformas

La siguiente gráfica recoge los datos de las ventas de videojuegos de plataformas a lo largo de su historia.

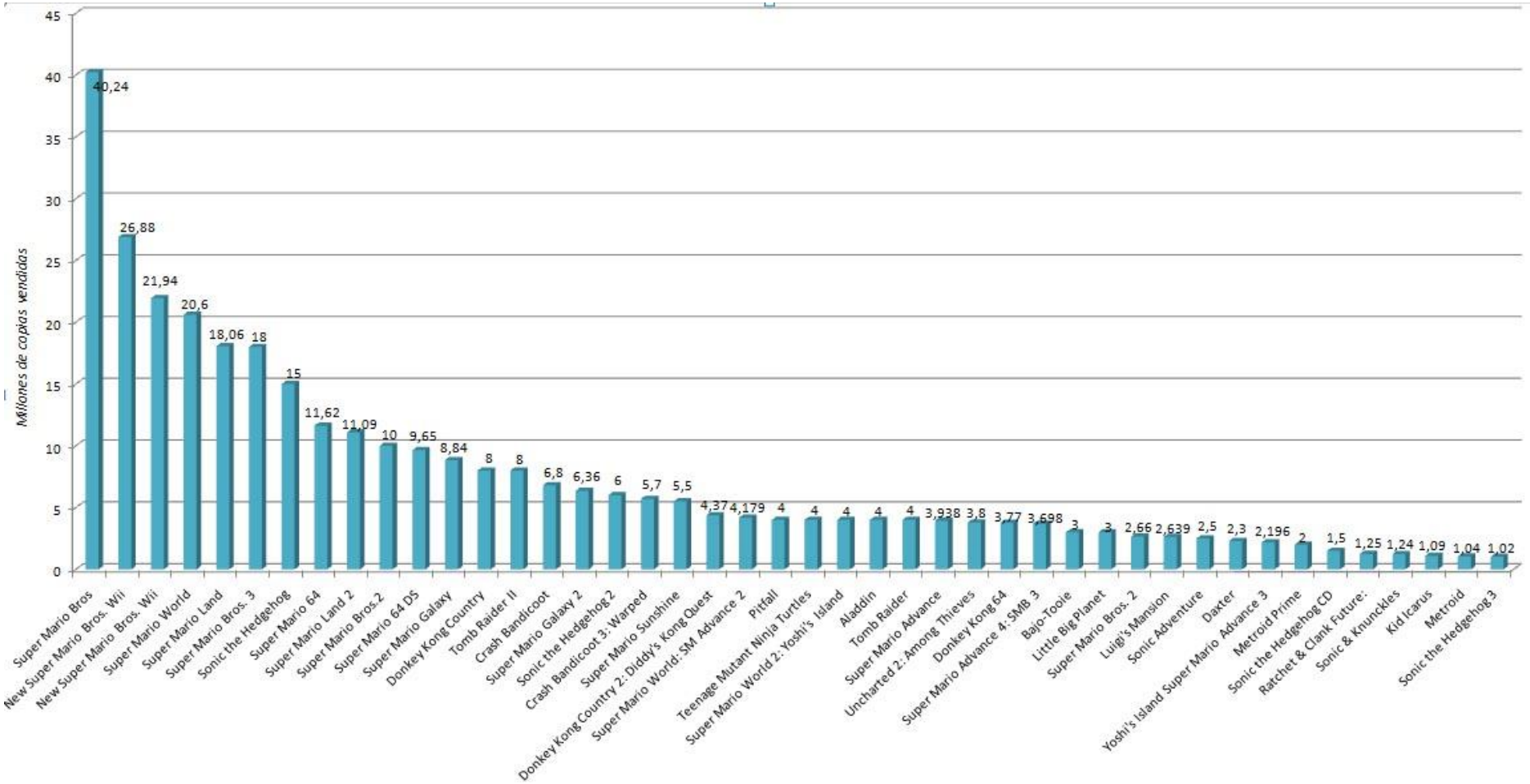


Ilustración 29 - Estadística venta videojuegos plataformas

2.4 Inteligencia Artificial en los videojuegos

2.4.1. Historia de la IA aplicada a videojuegos

La primera aplicación de la Inteligencia Artificial en el campo de los juegos se centra en los clásicos ajedrez y damas, así como juegos de lógica. En 1959 Arthur Samuel publica el artículo "Some Studies in Machine Learning using the Game of Checkers [56]. Se habla de la utilizaban las técnicas de búsqueda como el A*, o el algoritmo Mini-max para varios jugadores y tener en cuenta sus movimientos. En el siguiente artículo [57] se puede encontrar una referencia más detallada sobre la generación de árboles minimax.

Posteriormente, con la aparición de videojuegos como Space Invaders y Galaxian a finales de los años 70 se utilizaban funciones hash en función de las acciones del jugador humano [58].

En los años 80, se empieza a dotar de personalidad propia a los enemigos, para que cada uno actuase de una manera. Se refleja en juegos como Karate Champ o Pac-Man. En éste último, cada enemigo tenía su propia máquina de estados que le hacía comportarse con cierta tendencia propia.

Con una cultura del mundo del videojuego mucho más desarrollada, durante la década de los 90 se fueron sucediendo nuevas aplicaciones de técnicas, como por ejemplo Battlecruiser 3000AD añadió el sistema de redes neuronales. Estos métodos no se utilizarían en muchos juegos más, aunque el exitoso Colinc Mc Rae posterior es otro ejemplo que utiliza redes neuronales. También en los 90 surgen videojuegos como Creatures que utilizaría técnicas de aprendizaje por refuerzo.

En el género de los *shotter*, aparece Half- Life, el cual es un gran ejemplo de scripts que intentan aparentar ser una IA real, y con resultados bastante parecidos. Y siguiendo con el mismo género pero ya en pleno siglo XXI, van surgiendo títulos con una Inteligencia Artificial cada vez más mejorada. Halo (2001) se utilizan los denominados *behaviors tres,* una nueva herramienta para la IA a medio camino entre los planificadores y las máquinas de estados. Se conseguía dotar a la IA con la capacidad de utilizar vehículos para desplazarse y desarrollar sus acciones tácticas; así como el reconocimiento del tipo de amenazas y su reacción inmediata. En Far Cry (2004) los enemigos reaccionaban a relación del estilo de juego del jugador. F.E.A.R. hacía uso del sistema GOAP (Goal-Oriented Action Planning) para adaptarse a los compartimientos de forma dinámica. Por última, en el año 2008, el videojuego Left 4 Dead utilizaba la tecnología denominada *IA director* para mantener un nivel de tensión adecuado para el jugador, y que éste ni se aburriese ni se frustrase.

En la siguiente referencia [59] se puede consultar un artículo con los 10 juegos más influyentes de la historia en cuanto al campo de la Inteligencia Artificial se refiere. De algunos de ellos ya se ha hablado hasta el momento; así que se expone a continuación un resumen de los demás no tratados:

- SimCity - 1989 – Primer videojuego que proporcionaba el control de simulaciones muy complejas. El juego consistía en la creación, gestión y evolución de ciudades. A partir de un terreno vacío, el jugador tiene como objetivo crear una ciudad que prospere y llegue a convertirse en una gran urbe. Una característica curiosa del juego es que no se podía ganar ni perder. Además, cada partida era distinta a la anterior por el sistema de IA.

- Total War (2000) – Control de miles de agentes IAs. Modela también las emociones de los soldados según el estado de la batalla. Su lógica está inspirada en el clásico de la literatura "El Arte de la Guerra" de Sun Tzu. Destacaba, además por su potencial gráfico, por

su dinámica del juego, en la que se debía combinar estrategia por turnos y tácticas de batalla en tiempo real.

- Los Sims – 2000 – [60] Basado en su predecesor SimCity, aunque en este juego se centra en la simulación de los personajes en particular y no la ciudad en general. Es de los juegos del género simulador de vida más famoso e influyente. La forma de interactuar con los objetos se almacena en los objetos, pero la verdadera IA de este juego reside en cómo actúan los personajes (cuando deciden hacer algo y cómo aprenden). Destaca el cierto grado de libre albedrío con el que se dota a los personajes. Por ejemplo, cuando pasa un tiempo sin ordenar nada al personaje, éste toma sus propias decisiones. Aunque bien es cierto, que hay cierto conjunto de acciones esenciales que se le tienen que ordenar explícitamente, como por ejemplo el pago de facturas. Otro aspecto novedoso que incluye es la interrelación de unos personajes con otros, y la simulación de sensaciones al respecto. En el siguiente artículo [61] se muestran técnicas relacionadas con la IA que se han usado en el propio desarrollo del juego. Y en la siguiente referencia se puede encontrar una presentación con aspectos más técnicos de la programación de la IA de los Sims [62]. Tras el éxito arrollador del producto en el mercado, han ido saliendo muchísimas packs de expansiones de muy diferentes ámbitos, como por ejemplo mascotas, Medieval, etc. Del mismo modo, salió la segunda versión del título, con sus correspondientes expansiones, y la tercera (y última hasta el momento). El aspecto más novedoso de esta última entrega y lo que han potenciado más es la personalidad propia de cada personaje, como se puede apreciar en el siguiente artículo y el vídeo contenido [63]

Ilustración 30 - Los Sims

- Façade – 2005 – Interacción mediante PLN. El lenguaje provee formas de especificar el comportamiento de los personajes en términos de una historia dinámica.

- Black & White – 2001- Videojuego simulador de dios en el que el usuario ejerce, propiamente, el papel de *Dios* en el mundo que crea. Su género es una mezcla de estrategia y rol, pero incluye el cuidado de una criatura por aprendizaje. El jugador establece contacto con el mundo de Black & White y mediante el uso de una Mano va interaccionando con las aldeas de ese mundo.

2.4.2. IA en juegos de fútbol

Un ejemplo de gran uso y aplicación de Inteligencia Artificial es Otro género de los videojuegos donde se aplica la IA, y con especial importancia, son los simuladores deportivos. Pongamos como primer ejemplo el llamado *deporte rey,* el fútbol.

Y no sólo en el mundo de los videojuegos digital, sino también tiene una gran cabida en los robots. Existe una competición de ámbito mundial llamada RoboCup [64]. Su objetivo es promover las investigaciones en robots e inteligencia artificial con este deporte. Aparte de la competición mundial, hay otros eventos en otro ámbito más local. Por ejemplo, la web de la competición española es esta [65]. La Universidad Carlos III de Madrid ha estado muy implicada en estos proyectos por medio de su departamento CAOS [66] y existen varios Proyectos Fin de Carrera sobre este tema. En el siguiente artículo [67] se habla sobre la aplicación de enseñar Inteligencia Artificial a ingenieros a través de Robocup. En la siguiente referencia [68] se puede consultar una presentación sobre la IA multiagentes y la robocup.

Ilustración 31 - Robocup

Pero, aparte de estos sistemas de robots jugadores de fútbol, obviamente también tiene su repercusión en las videoconsolas. Los dos máximos exponentes en la actualidad de este tipo de juegos son FIFA, de la compañía EA Sports; y Pro Evolution Soccer, de la nipona Konami. Obviando el apartado gráfico, donde cada vez se están logrando resultados más fidedignos a la realidad ayudados por la tecnología de las nueva generación de consolas, el aspecto de la jugabilidad (y por consiguiente la inteligencia artificial) va ganando posiciones para convertirse en piedra angular del éxito de un producto. Como se muestra en la referencia anterior de los gráficos de los juegos, se está llegando a un punto de realismo tan alto que, el superarlo va siendo demasiado complicado. Ante tal competitividad, el factor de la IA pasa a ser referente para la elección de uno u otro producto. Y en este campo, los avances también van siendo cada vez más impresionantes. En el siguiente artículo [69] se puede observar cómo cada compañía tiende sus esfuerzos de la IA según su estrategia: FIFA tiene a mejorar la IA con respecto a nuestros adversarios, mientras que PES 2012 intenta mejorar la actuación de nuestros compañeros de equipo.

Ilustración 32 - IA PES 2012

Siguiendo con el tema, a continuación se muestra otro ejemplo concreto sobre el nuevo sistema de IA para FIFA 12 denominado Personality Plus. Éste se caracteriza, entre otros muchos aspectos, en cómo reacciona el jugador agente del ordenador para realizar los pases entre los miembros del equipo. En el siguiente enlace [70] se puede ver un vídeo de ejemplo donde se muestra que en función de las características del delantero, se realice un centro al área con un pase alto o un pase bajo. Si el delantero a rematar es Peter Crouch se centrará a su cabeza, mientras que si es David Villa, el pase irá a ras de suelo.

2.4.3. Dificultad y jugabilidad relacionadas con la IA

La inteligencia Artificial en los juegos es importante porque afecta directamente a la jugabilidad de éste. Sin embargo, a la vez se convierte en un tema controvertido. El objetivo final de un videojuego es entretener al usuario y que pase un buen rato. Para ello, ha de tener unos niveles de jugabilidad óptimos. Esto significa que ni la Inteligencia Artificial del ordenador sea excesivamente perfecta, porque siempre ganaría todas las partidas y el usuario, obviamente, se frustraría; ni que sea excesivamente fácil, pues el jugador se aburriría. Este factor de dificultad es totalmente subjetivo al usuario en concreto: si es un jugador experimentado; o, en cambio, es un jugador ocasional. Muchos videojuegos lidian con este aspecto generando varias dificultades para un mismo juego, el cual el usuario puede modificarlas según sus propias características que él

considere, por ejemplo el *God of War* de PlayStation 2. Otros, en cambio, no permiten la selección de dificultad al usuario, sino que van aumentándola a medida que el juego va avanzando, como por ejemplo el Super Mario Bros. Es importante destacar que, no toda la dificultad de un juego reside en la IA de éste (puede haber juegos que no tengan, por ejemplo, o elementos fijos del mapa); pero, en cambio, la Inteligencia Artificial de un juego siempre influirá en los niveles de dificultad. En el siguiente enlace [71], se muestra un video recopilatorio con diez de los videojuegos más difíciles de la historia. Destacan en los dos primeros puestos los juegos de plataformas: Ghouls & Ghost y Battletoads.

Ilustración 33 - Ghost & Goblins

2.4.5. Uso IA con mayor hardware

Otro aspecto importante es que con las nuevas generaciones de consolas, en la que cada vez es mayor el potencial computacional del hardware, se tienen más recursos disponibles para el uso de la IA. Antiguamente, con máquinas tan limitadas, una gran parte del peso computacional del juego iba al multimedia. Y las plataformas fueron creciendo con esta óptica siempre en mente. Pero fue llegando un momento en que "sobraba capacidad", y esto es el escenario ideal para explotar al máximo la inteligencia artificial. Un claro ejemplo de esto es el chip Cell [72], que se utiliza actualmente en la Play Station 3.

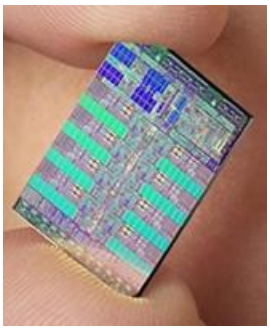


Ilustración 341 - Chip Cell

No hay que olvidar en ningún momento que estas técnicas requieren mucho cómputo. Por ejemplo, el algoritmo A* que se utilizará en este proyecto, por sus características tiene una facilidad impresionante para utilizar memoria. Esto puede ocasionar, como es lógico, que la memoria se llene; y hay que evitar siempre estos hechos para que la aplicación funcione correctamente. Estas circunstancias han hecho que las optimizaciones de código cobren especial importancia.

En el artículo publicado [73] por el diario británico The Guardian en 2007, se afirmaba que los juegos habían aumentado enormemente sus capacidades gráficas; pero, en cuestiones de Inteligencia Artificial, seguían teniendo prácticamente la misma tecnología que hacía nada menos que 10 años. Para afirmar tal circunstancia, se basaba en las palabras de Peter Molyneux, prestigioso diseñador y programado de juegos. Es creador de grandes éxitos de videojuegos como Theme Park, Theme Hospital, Fable o Black & White. Todos estos juegos se caracterizan por tener un gran uso de la Inteligencia Artificial; y actualmente trabaja como responsable del proyecto Kinect de Xbox 360 integrando la IA. Peter sostenía que había que diferenciar entre la Inteligencia Artificial pura, y la que se estaba implantando en los videojuegos.

Otro prestigioso investigador, Steve Grand afirmaba en el mismo artículo que, hasta la fecha, la gran mayoría de videojuegos no disponían de una Inteligencia Artificial como tal, sino que estaban compuestas básicamente de sencillas secuencias de programación IF / THEN intentando controlar todos los estados y posibilidades del juego; pero sin un planteamiento sólido desde la perspectiva de la IA. Por tanto, se podría

decir que eran sistemas expertos y en ellos se basan los inicios de la IA. Otro ejemplo que ponía es que los personajes no aprenden por sí solos, premisa básica para decir que un agente es inteligente, sino que simplemente hace lo que el programador decide en cada situación pre-estipulada. Steve es el creador del videojuego Creatures [74] en el cual hay que enseñar a una criatura a que hable, coma, y se defienda por sí sólo y vaya evolucionando. Es de los pocos juegos que se puede decir que tienen un sólido pilar de Inteligencia Artificial pura como pilar fundamental.

Ilustración 35 - Creatures

No obstante, no cabe duda que el desarrollo de la inteligencia artificial aplicada a los videojuegos crece rapidísimamente. Como muestra de ello, es el siguiente artículo [75] en el que narra cómo un grupo de investigadores del MIT ha conseguido enseñar a un ordenador a jugar al juego Civilation sólo proporcionándole el manual del juego. Además, el experimento fue todo un éxito, ya que consiguieron que el ordenador ganase a los jugadores humanos en un tasa del 79 % de las partidas.

2.4.6. Técnicas IA para videojuegos

Existen muchísimas y diferentes técnicas de Inteligencia Artificial a aplicar en los videojuegos; pero no es el objetivo de este estudio el detalle

de todas y cada una de ellas. Sin embargo, se expondrán aquí algunas de las más importantes así como un resumen de lo más esencial:

- Pathfinding – Algoritmos de búsqueda. Su finalidad de encontrar el camino óptimo entre dos puntos. Muy usado también en terminales de geoposicionamiento (GPS). Ejemplos de esta técnica son el algoritmo A* (utilizado en este proyecto) o el algoritmo de Dijkstra.

- Máquinas de estados – Muchos juegos implementan esta técnica en la que, existe cierta controversia sobre si es Inteligencia Artificial *pura*, o no. El debate converge en si el algoritmo puede ser catalogado como técnica de IA ya que sólo actúa ante estados predeterminados. Sin embargo, es innegable su uso que se ha dado en la historia de los videojuegos para estas finalidades.

- Aprendizaje por refuerzo – Técnicas que se basan en enseñar a personajes a través de acciones, que irán teniendo repercusiones en el futuro [76]. Se premian las acciones buenas y se castigan las malas.

- Inteligencia estafadora - *Rubber-bandrule.* Se utilizar para detectar si el jugador humano tiene una situación mucho más ventajosa que el ordenador, y actúa en consecuencia. Juegos que lo utilizan son el EA Sport's Maden NFL o el Super Mario Karts.

2.4.6.1. Máquina de estados

Es una de las primeras técnicas que aparecieron en los videojuegos, aunque se siguen usando en la actualidad, debido a su fácil implementación. Una máquina finita de estado es una abstracción

compuesta por un conjunto de estados y de transiciones entre éstos. Dichas transiciones se efectúan al producirse ciertas condiciones desde el exterior que afectan al estado en cuestión. Ante tales circunstancias, la máquina genera unas acciones que le hacen cambiar su estado actual por el correspondiente.

Para diseñar una máquina de estados hay que analizar todas las situaciones y comportamientos consecuentes que podrían producirse; y modelizarlo a partir de ellos.

2.4.6.2 Búsqueda de Caminos

El objetivo que se persigue es que el personaje sea capaz de desplazarse por el escenario del juego con cierta *conciencia propia*. Es decir, que el personaje sea capaz de detectar el mundo en el que está, analizarlo, y decidir cuál sería el mejor camino para llegar a un objetivo conocido.

Existen varios algoritmos para conseguir tal objetivo:

- Crash and Turn – Intenta avanzar siempre en línea recta hacia el destino. Si se encuentra un obstáculo, gira hacia la izquierda o la derecha hasta que tenga "visión" del objetivo y reemprender la marcha recta.

- Algoritmo de Dijkstra – Muy útil si se puede representar el mundo o escenario en forma de grafo. De esta manera, a cada zona se le asigna un vértice, y a las aristas que les une un peso, para poder ejecutar el algoritmo.

- A* - Es el método utilizado en este proyecto. Si existe el camino, siempre encuentra el más corto. Se explicará en detalla en el apartado de implementación.

Capítulo 3

Descripción del videojuego

3. Descripción del juego

3.1. Descripción textual del juego

SuperEnjutoIA* es un juego de plataformas en dos dimensiones. El scrolling (desplazamiento del nivel) es horizontal.

La modalidad de juego es individual (un solo jugador); pero, se compite contra el ordenador, en el mismo nivel y en el mismo tiempo. El usuario maneja un personaje, y el ordenador controla, mediante técnicas de inteligencia artificial, otro distinto. Ambos buscan un mismo objetivo, maximizar la puntuación.

Por tanto, el objetivo del juego es conseguir los máximos puntos posibles en el nivel antes que finalice el tiempo. El jugador que más puntos consiga resulta ganador. Estos puntos se pueden conseguir de dos maneras:

- Recogiendo *monedas*: Son objetos repartidos a lo largo del nivel. Hay tres tipos de monedas, cada una con una puntación distinta:

    - Moneda Amarilla – 250 ptos.
    - Moneda Verde – 500 ptos.
    - Moneda Roja – 1000 ptos.

  El número de monedas de cada tipo en el nivel dependerá de su puntuación, es decir, habrá más monedas amarillas que verdes, y más que rojas.

- Superando el nivel lo antes posible El nivel tiene un contador de tiempo; el cual marca los límites de la duración de la

partida para que el jugador alcance la meta. Cada nivel tiene su duración propia. El contador se irá decrementando en una unidad cada segundo. El número de puntos que el jugador obtiene al superar el nivel se calcula en función del tiempo que le ha sobrado. Por cada segundo que no haya utilizado, conseguirá 50 puntos. Por tanto, cuanto antes consiga superar el nivel, más puntuación obtendrá.

Cada nivel tiene un único punto de inicio de partida, y un único punto de meta; tanto para el personaje del jugador humano como para el personaje controlado por el ordenador.

Un nivel se compone también de un conjunto de *enemigos*. Su objetivo es causar daño a los personajes que intentan llegar a la meta. Existe un único tipo de enemigo:

- Enemigos Bill – Su movimiento es únicamente horizontal. Cuando encuentra un obstáculo en su camino, cambia el sentido de su movimiento, y así indefinidamente.

Un enemigo causa daño a los personajes al tocarles. Cuando esto ocurre, el personaje pierde monedas. El número máximo de monedas que puede perder son 12 (6 amarillas, 4 vedes y 2 rojas). En caso de que no tuviese suficientes monedas de algún tipo, esa diferencia se añadirá al número de monedas que se tiren del siguiente tipo. En caso de no tener suficientes monedas, arrojaría todas y se quedaría a 0.

Las monedas de penalización se reparten al lado del personaje durante 5 segundos para que pueda recogerlas si quiere. Cuando estos cinco segundos se pasan, las monedas desaparecen.

Si el personaje humano se queda sin monedas, la partida acaba. El personaje del ordenador no influye para que una partida acabe. El otro

evento que puede ocurrir para que una partida acabe es que se agote el tiempo.

Los movimientos básicos del personaje son el desplazamiento hacia la izquierda, hacia la derecha y el salto. También puede saltar hacia ambos lados combinando movimientos.

El usuario y el ordenador controlan a su personaje al mismo tiempo, pero en distinto plano. Lo que quiere decir esto es que no es necesario que ambos personajes estén en la misma parte del escenario. Cada uno puede jugar su partida como desee. Sin embargo, la cámara que va siguiendo el desarrollo del juego mostrará en todo momento al personaje del usuario. Si el personaje del ordenador estuviese también en una posición dentro de lo que recoge la cámara, también se mostrará por pantalla.

La morfología de los niveles consta de una *base* y distintas *plataformas.* La base está siempre durante todo el nivel. Las plataformas pueden aparecer a lo largo del todo nivel, y a distintas alturas. Para acceder a las plataformas, el personaje tendrá que saltar para poder alcanzarlas. Si se tira de dichas plataformas, el personaje cae a la base.

El usuario podrá crear sus propios niveles (así como especificar la duración del tiempo de los mismos) por medio de sencillos ficheros de texto plano.

En la siguiente imagen se muestra un esquema simplificado de la morfología de un nivel de ejemplo.

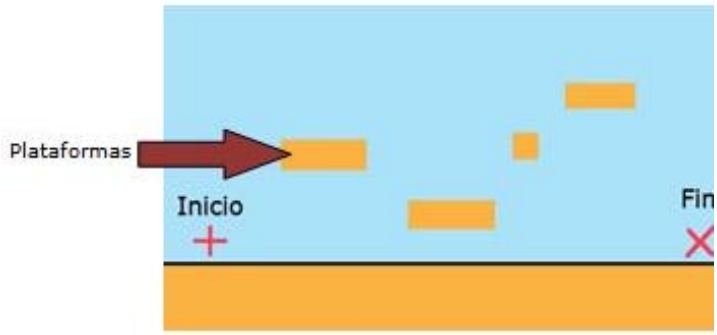


Ilustración 36 - Ejemplo de nivel simplificado

3.2. XNA

3.2.1. Microsoft XNA

Microsoft XNA [77] (XNA is Not anAcronym) es un conjunto de herramientas, utilidades y librerías con el fin de facilitar el desarrollo de videojuegos para las plataformas Windows, Xbox 360, Zune y Windows Phone (XNA 4.0).

XNA (XNA's Not Acronymed) es un conjunto de herramientas destinadas a la creación de videojuegos. La empresa desarrolladora de XNA es Microsoft. Las plataformas para las que se pueden desarrollar dichos juegos son Windows, Xbox 360, Zune y, en su última versión 4.0, Windows Phone 7. El objetivo de XNA Framework es la simplificación de las tareas para la creación de un videojuego utilizando su API. Un ejemplo claro es la programación de los gráficos del juego y su relación con DirectX. Se puede usar la API de XNA para tales efectos y, por tanto, no es necesario usar las instrucciones específicas de DirectX. Lo mismo ocurre con el sonido, la entrada de datos, etc. En definitiva, XNA es un framework [78] que simplifica la programación de videojuegos.

El framework XNA está basado en el framework 2.0 para Windows y en el Compact Framework 2.0 para Xbox 360. Por tanto, también se ejecuta sobre Common Language Runtime, pero con una versión de ésta optimizada para videojuegos. CLR [79] es un entorno de ejecución para los programadores de la plataforma Microsoft .NET. Su objetivo es compilar un código intermedio denominado Common Intermediate Language (CIL) a código máquina. Aunque es parecido en cierto modo a una máquina virtual, no ha de confundirse con ésta, ya que CLR se ejecuta nativamente sin intervención de una capa de abstracción sobre el hardware. Por tanto, un compilador convierte en tiempo de compilación el código escrito en lenguaje .NET (C# o VB .NET) a código CIL. El CLR es el encargado de convertir, en tiempo de ejecución, dicho código CIL a código nativo del sistema operativo. De esta manera, el programador consigue abstracción sobre muchos detalles específicos. Se muestra todo este proceso en el siguiente diagrama:

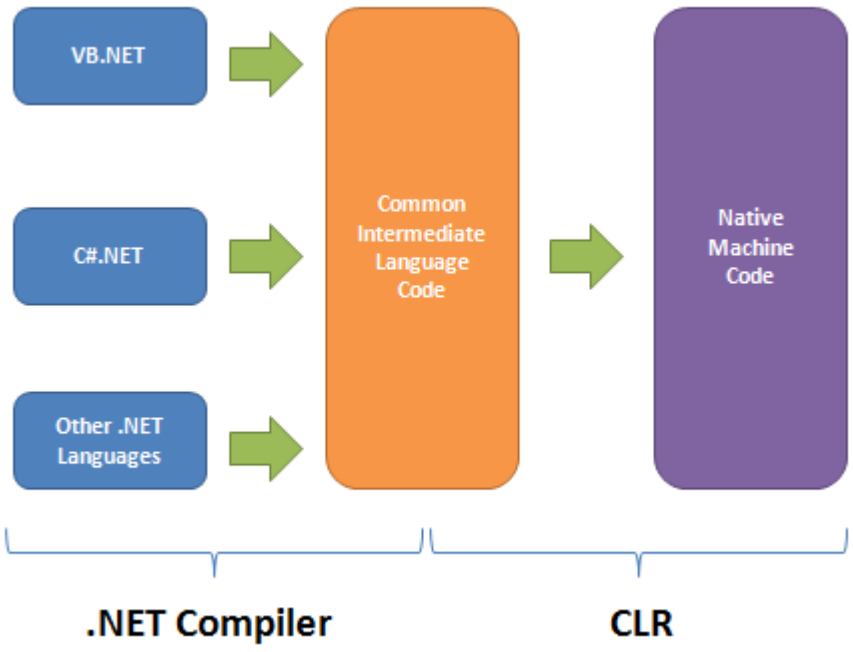


Ilustración 37 - CLR

El desarrollo en XNA está basado y orientado a componentes y contenedores. Un componente es un fragmento de código que siempre cumple la misma funcionalidad. De esta forma, se tiene la gran ventaja de la reusabilidad y una capa de abstracción que facilita la programación. Por

ejemplo, el componente que dibuja por pantalla o que se encarga de cargar los recursos. Es importante destacar que XNA no obliga a hacer todo el programa basado en componentes, sino que es una opción que ofrece., ya que el usuario puede crear sus propios componentes. Por su parte, un contenedor es una agrupación de componentes.

Todos los GameComponent tienen sus propios métodos de Inicializar (Initialize), Cargar (Load), Descargar (Unload), Actualizar (Update) y Dibujar (Draw). La clase principal del juego (Game) se encarga de gestionar todos los demás GameComponents que forman la aplicación.

Estos métodos nombrados anteriormente, son la base del mecanismo de funcionamiento de XNA. Al cargar un nuevo componente, lo primero que se produce es la llamada a su método *Initialize.* Este se encarga de inicializar la clase, y así como asignar algunos valores específicos, como por ejemplo la resolución de pantalla. Posteriormente, se llama al evento *LoadContent*, el cual es el encargado de cargar los recursos que necesite dicha clase, como por ejemplo los sprites o las fuentes. A partir de ese momento, el componente ya ha sido cargado; y empieza su el ciclo básico de ejecución. Éste consta dos componentes principales: *Update* y *Draw.* El primero es el encargado de llevar toda la lógica del juego, como por ejemplo los estados del mundo y de los personajes. El segundo, es el encargado de pintar en la pantalla. Es importante destacara cómo funcionan entre sí ambos componentes: Draw se ejecuta un número fijo de veces por segundo (generalmente suelen ser valores entre 24 y 30 frames por segundo). El tiempo de reloj que sobre en cada segundo es el tiempo que se le asigna al método Update. Este tiempo se puede fijar por el desarrollador, mediante las propiedades TargetElapsedTime y IsFixedTimeStep.; ó, por el contrario, que sea XNA el encargado de gestionar automáticamente estas transiciones. Por tanto, el programador no se tiene que preocupar de estas llamadas entre métodos porque es el propio motor de XNA el que se encarga de ir ejecutando el ciclo entre ambas.

A continuación se muestra un esquema de su modo de funcionamiento:

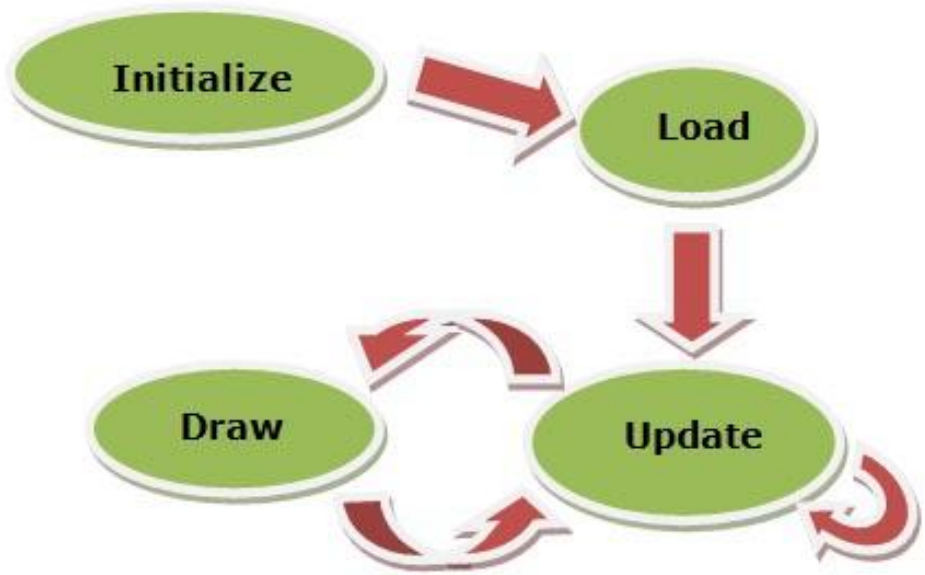


Ilustración 38 - Esquema funcionamiento XNA

3.2.1.1 Versiones

A fecha de hoy, existen 4 versiones publicadas. A continuación se detallará cada una de ella:

- XNA Game Studio Express – Fue la primera versión en salir a la luz, en agosto del año 2006. Se sacó una actualización llama XNA Game Studio Express 1.0 Refresh en la que sí se permití a compartir los juegos con otros usuarios de Xbox3 60, ya que la anterior no lo permitía.
- XNA Game Studio 2.0 – Diciembre de 2007. Añade librerías para los juegos en red.
- XNA Game Studio 3.0 – Septiembre 2008. Permite el desarrollo de juegos para la plataforma Zune [80]. También incorpora nuevas funcionalidades para el juego en red, en especial para Xbox LIVE.

- XNA Game Studio 3.1 – Junio de 2009. Nueva API con soporte de vídeo; y revisión de las antiguas para la inclusión de los avatares en Xbox 360.
- XNA Game Studio 4.0 – Es la versión actual; fue lanzada en Septiembre de 2010. La principal novedad es que incluye desarrollo para Windows Phone 7; e integración con la última versión del IDE de programación (Visual Studio 2010). A continuación se comentarán más detalles de esta versión, ya que es la usada para la elaboración del proyecto.

Como se ha comentado anteriormente, la principal novedad de la última versión de XNA es su integración de Windows Phone 7. Este hecho es muy significativo por la era de los móviles en la que vivimos; y la gran batalla que se está librando en el mercado por ella. Los grandes aventajados son Apple con su iPhone y el sistema operativo libre Android, seguido por Blackberry. Un ejemplo de esto, es el estudio realizado por la importante agencia Nielsen sobre el mercado de los Smartphone en Estados Unidos [81].

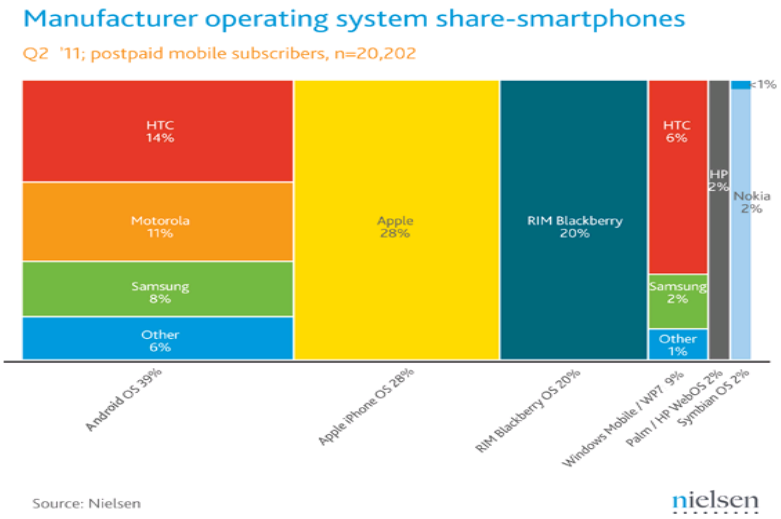


Ilustración 39 - Tasa de mercado de móviles

Sin embargo, Microsoft, evidentemente, no ha querido quedarse fuera. De las primeras determinaciones que tuvieron fue la unión [82] con la compañía de móviles sueca Nokia. Ésta había perdido con diferencia su antigua posición de líder en móviles en el mercado, coincidiendo con el

boom de los Smartphone, ya que su sistema operativo Symbian [83] no había estado a la altura.

3.2.1.2. Novedades técnicas

Con la actualización a la versión 4.0 se incluyeron muchísimas novedades y cambios, muchas más que en cualquiera de las actualizaciones anteriores. En esta referencia [84] al soporte oficial de Microsoft se puede ver la lista completa de cambios. A continuación, se detallarán aquí las más significativas:

- Desarrollo de juegos para Windows Phone 7.
- Interfaces gráficas simplificadas.
- Efectos configurables.
- Mayor soporte de Audio
- Enumeraciones de música, imágenes y playback video.
- Soporte de la pantalla táctil de los móviles (Framework.Input.Touch).

Con todas estas novedades, muchas de las técnicas para la realización de tareas han cambiado. En la siguiente referencia [85] se encuentra un práctico resumen de las diferencias de programación entre versiones que existen para una misma funcionalidad. Por último, en este blog [86] también se pueden encontrar distintos artículos sobre las diferencias de esta nueva versión.

3.2.1.3. Scene XNA

Aparte de las novedades oficiales de Microsoft, van saliendo proyectos alternativos de desarrolladores independientes que aumentan aún más las posibilidades que ofrece XNA. Un buen ejemplo es XNA Touch [87], el cual permite ejecutar código XNA sobre plataformas MONO [88]. Cobra especial interés porque estas otras plataformas pueden ser Android o iPhone. Es decir, XNA Touch permite ejecutar el código directamente, sin traducción,

sobre otras plataformas "No-Microsoft". Para ello, hace uso de la tecnología OpenGL [89].

Ilustración 40 - Ejemplo XNA en iPhone

De momento el proyecto cuenta con algunas "carencias" o "requisitos especiales" que no lo hace totalmente accesible a la comunidad desarrolladora. Por ejemplo, es necesario disponer de un MAC, por lo cual los desarrolladores que trabajen en entorno Windows no podrían utilizarlo. Además, para disponer de todas las funcionalidades completas se necesita una licencia de pago, aunque bien es cierto que se dispone también de licencia de prueba (trial) para su descarga. Otro ejemplo de este aspecto es que no dispone todavía de soporte para 3D.

No obstante, y a modo de resumen, éste es un interesantísimo proyecto el cual es recomendable seguir de cerca por su gran proyección y las posibilidades que podrá ofrecer. A continuación se muestra una referencia [90] a un tutorial de cómo hacer una pequeña aplicación utilizando esta tecnología.

Otro ejemplo muy significativo del uso potencial de XNA es el relacionado con Kinect. Kinect es un periférico, lanzado en principio para Xbox 360, el cual es capaz de detectar los movimientos del jugador con su cámara integrada. Así se consigue la gran experiencia para el jugador de

controlar todo el videojuego sin utilizar ningún mando, simplemente utilizando su cuerpo.

Ilustración 41 - Kinect

Se dijo anteriormente que el periférico fue lanzado en principio únicamente para Xbox 360. Era la única plataforma que los soportaba, no existían drivers para PC. Sin embargo, este hecho cambió con una curiosa actividad propuesta por la empresa Industrias Adafruit. Esta propuesta se basaba en dar una recompensa de 3000 dólares a la primera persona o grupo que descubriese un driver open-source para el dispositivo [91]; resultando ganador [92] el español Héctor Martín [93] mediante ingeniería inversa , hacker bastante conocido en la scene [94]de los videojuegos por sus constantes aportaciones desde su web [95].

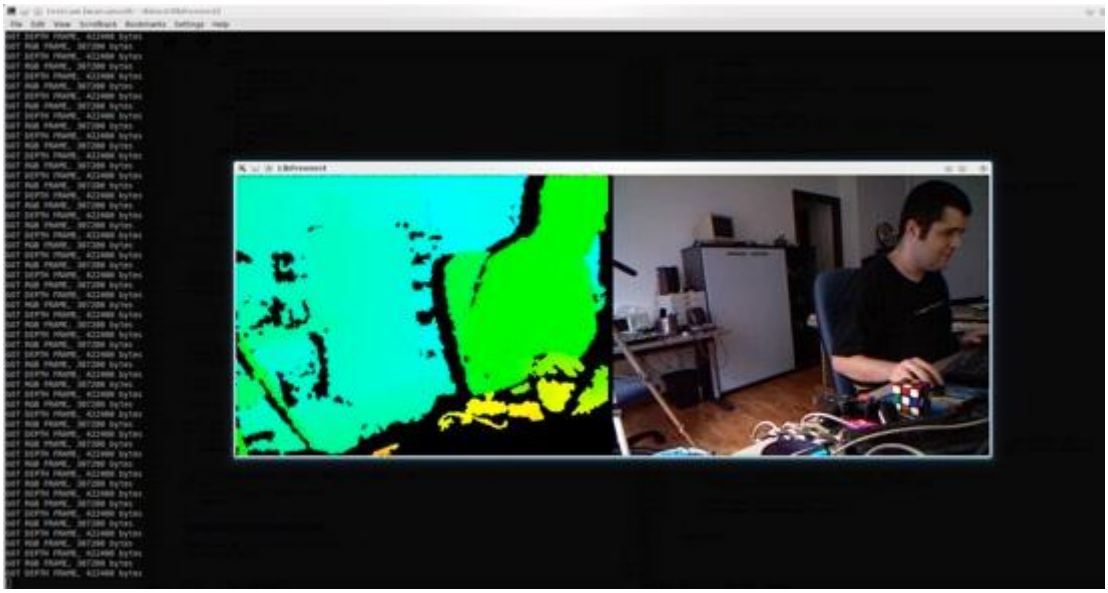


Ilustración 42 - Driver Kinect

Ante esta situación, Microsoft decidió liberar los drivers para su uso en cualquier plataforma; y también el kit de desarrollo. En la web oficial del

SDK [96] se pueden descargar dichos drivers, así como su documentación [97].

Aunque oficialmente no hay soporte todavía por parte de Microsoft para Kinect con XNA, en un futuro inminente llegará, según palabras del propio director de proyecto de Kinect [98]:

"hemos tenido, tenemos y seguiremos teniendo herramientas gratuitas que la gente puede descargar - el XNA Studio - y que permiten crear aplicaciones para Xbox 360. Ahora mismo no hay soporte Kinect en XNA, pero es algo a lo que daremos soporte en el futuro".

No obstante, la comunidad de desarrolladores ya empezó a trabajar en ello, como muestran las siguientes referencias [99] [100] [101] de Kinect con XNA.

3.2.1.4. Ejemplos de juegos

En la siguiente referencia [18] enlace se puede encontrar todo el catálogo que existe de juegos indie ofrecidos en el bazar del Xbox Live. Todos ellos han sido desarrollados por personas individuales o pequeños grupos de programadores en XNA. Dichos juegos son venidos a través de esta plataforma, repartiéndose proporcionalmente los beneficios los desarrolladores y Microsoft.

Además, se organizan competiciones de desarrollo de juegos en XNA. Una de las más famosas es Dream.Build.Play, con importantes premios. La siguiente referencia es la página web oficial del evento [102]

Algunos ejemplos de juegos, desarrollados por españoles, que han conseguido bastante éxito son los siguientes:

- Rotor Scope – [103] – Juego de habilidades e ingenio; que además cuenta con un interesante hilo argumental.

Ilustración 43 - Rotor Scope

- Kaotik Puzzle – Juego de puzles del estilo a Tetris que permite competir contra otros jugadores, o bien contra el motor de Inteligencia Artificial propio del juego.

Ilustración 44 - Kaotik Puzzle

3.2.1.5. Ventajas

Las principales ventajas que ofrece la utilización de XNA son:

1. Entorno de programación simple y sencillo de utilizar.
2. Ahorro de tiempo y eficiencia en la creación de contenedores y dispositivos gráficos.
3. Facilidad para la importación y carga de gráficos (imágenes o modelos 3D) en el proyecto.
4. Entorno para la importación y edición de audio.

5. Api específico para los cálculos matemáticos; así como para la gestión de colisiones.
6. Buena curva de aprendizaje.

3.2.2. Alternativas a XNA

Adventure Game Studio – [104] Orientada al desarrollo de aventuras gráficas, género que se hizo muy popular gracias a juegos como Monkey Island o Maniac Mansion. La última versión es la 3.2.1. No son necesarios conocimientos previos de programación, por lo cual el programa es más accesible para más público. Sin embargo, para los programadores también se ofrecen herramientas de scripting para programar en mayor profundidad el juego. Para dichos scripts, usa la metodología de programación orientada a objetos (POO). Además, gracias a estos script, se puede llegar a realizar juegos de otros estilos como por ejemplo RPG. Las ventajas de esta herramienta es que es para tanto un público experimentado como novato, y ofrece un potencial acorde para ambos sectores.

Allegro – [105] - Es una librería para el desarrollo de videojuegos. Fue desarrollada originalmente para Atari, y de ahí deriva su nombre (Atari Low-Level Game Routines).Es importante destacar que Allegro no es un motor de juegos, ya que puedes diseñar y programar tu juego como quieras. Ofrece librerías para facilitar la gestión de ventanas, la entrada de usuarios o el tratamiento de imágenes, por ejemplo.

Build Engine – [106] Motor de juegos de género shotter. Representa el mundo de modo bidimensional usando una malla denominada sectores con tal fin. Sin embargo, se renderiza de tal manera que parezca en 3 dimensiones. Esto tiene el efecto y la limitación de que la perspectiva depende de la distancia a la que se mire, y hay zonas las cuales no son visibles, como por ejemplo la parte de arriba de los edificios.

3D Game Studio – [107] - Es una herramienta comercial para el desarrollo de videojuegos en 3D. Incluye aplicaciones para la creación de niveles, gestión de scripts y colecciones de texturas y plantillas. Se puede programar con un lenguaje propio denominado Lite-C o con entornos de desarrollo externos de C++. Está orientado a distintos tipos de usuarios según sus conocimientos, desde principiantes hasta gente que se dedica profesionalmente al sector. La última versión es la A8, lanzada en 2010. Soporta imágenes y texturas de alta definición, así como acepta modelos desarrollado en otros entornos específicos de 3D, como por ejemplo Blender o 3ds Max.

GameMaker – [108] - Entorno de de desarrollo para Windows y MAC orientado a la creación de videojuegos. La última versión disponible para Windows es la 8.1; y para Mac la versión 7. Se caracteriza por permitir la creación de videojuegos a usuarios que no tengan altos conocimientos de programación. Aunque inicialmente empezó siendo un programa orientado a los juegos en 2 dimensiones, ha crecido abarcando cada vez más el modelo tridimensional, usando las librerías de DirectX y Direct3D. Tiene proyectadas la ´salida al mercado de nuevas versiones para Android y HTML5.

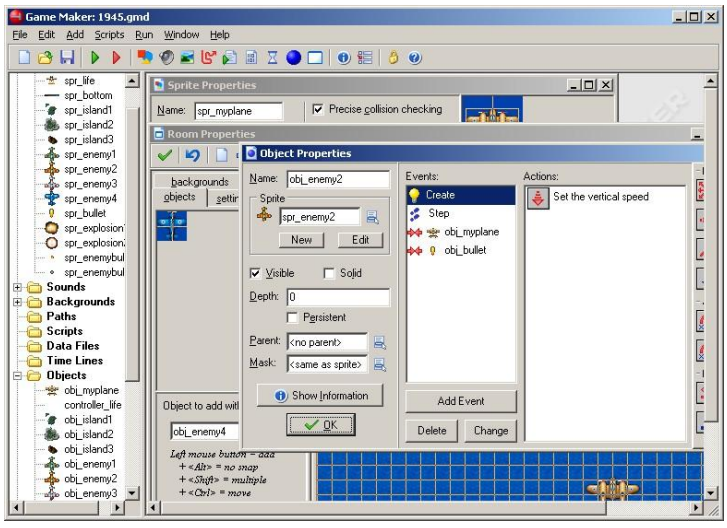


Ilustración 45 - Game Maker

Havok Game Dynimacs SDK – [109] - Motor físico utilizado en videojuegos para simular interacciones entre elementos. Entre sus funcionalidades incluye la detección de colisiones o la simulación de gravedad. En sus últimas versiones se utiliza la GPU en lugar de la CPU para su procesado. Se le puede considerar como una API especializada en la simulación del apartado físico, y ha sido utilizada en multitud de juegos comerciales por su potencia, como por ejemplo Age of Empires III, Assassin's Creen, Counter-Strike: Source, Portal 2 ó Fallout 3. Como se puede apreciar por los juegos, es utilizado en géneros de muy diferentes tipos.

LWJGL – [110] - Lightweight Java Game Library - Herramienta destinada a la creación de videojuegos escritos en lenguaje Java. Da acceso a tecnologías como OpenGL y OpenAL, pudiendo realizar juegos de alta calidad. El objetivo que persigue esta aplicación es permitir el acceso al desarrollo de videojuegos a la comunidad de desarrolladores de Java. Todas sus funcionalidades están integradas en una única API para facilitar su utilización.

M.U.G.E.N. – [111] - Motor de videojuegos de lucha en dos dimensiones de licencia libre y gratuito; y no se puede utilizar para fines comerciales y con ánimo de lucro. Ha tenido muchísimo éxito entre la comunidad desarrolladora de este género, por su grandísima facilidad de uso y los muchos recursos que existen para la aplicación. Se han sacado muchísimos ports de juegos comerciales con funcionalidades desarrolladas; así como "remixes" entre distintos juegos, como por ejemplo Street Fighter o Marvel vs. Capcom. La originalidad y creatividad son aspectos primordiales en este tipo de desarrollos.

Pygame – [112] - Colección de librerías SDL para la creación de videojuegos en 2D. Está programada en Python y orientado al manejo de sprites. La gran ventaja de este producto es la facilidad de uso y su curva de aprendizaje. Puede usarse también para programas multimedia o interfaces de usuario.

RPG Maker – [113] - Orientada a la creación de videojuegos de rol. Trae contenidos por defecto (gráficos, música, tipografía de fuentes…) para incluirlos en el juego, aunque se pueden importar nuevas que al diseñador requiera. Se han sacado varias versiones de la aplicación; siendo la última RPG Maker VX. Posteriormente salió una versión diferente denominada Action Game Maker que permite también crear juegos de plataforma y shotter. Una de sus ventajas es la amplia comunidad de desarrolladores que existen para esta plataforma. Por ejemplo, se muestra en la siguiente referencia, la comunidad española sobre el juego: http://rpgmaker.es/

Adventure Maker - http://www.adventuremaker.com/ - Aplicación para desarrollar juegos "point-and-click" para Windows, PSP o iPhone. No es necesario lenguajes de programación.

3DAdventure Studio – http://3das.noeska.com/default.aspx - Herramienta orientada al desarrollo de aventuras gráficas en 3 dimensiones. Está desactualizada ya que su última versión data del año 2004.

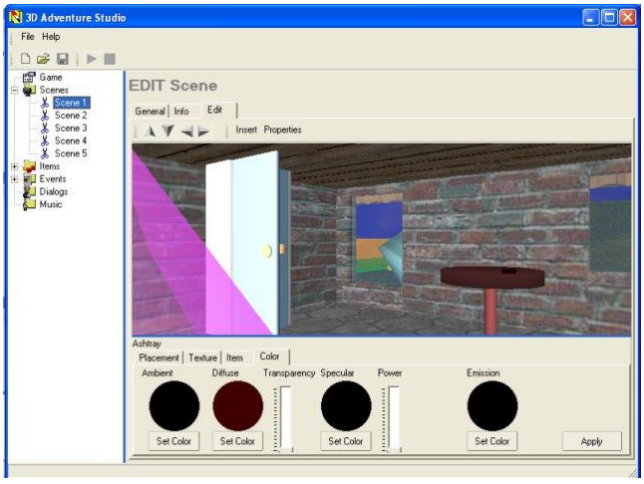


Ilustración 46 - 3DAdventure Studio

AGI Studio – [114] - Adventure Game Interpreter -Destinada a la creación de juegos con "estética clásica" o, como se le conoce en el mundo de los videojuegos, de "vieja escuela". Fue el punto de inicio de muchos juegos de aventura clásicos de la compañía Sierra, como los míticos Space Quest o King's Quest.

Ilustración 47 - Agi Studio

Visionaire – [115] - Herramienta orientada al desarrollo de juegos del género aventura gráfica. Más novedoso y actual; pero es de pago, aunque tiene una versión de prueba. Orientado a los gráficos de alta definición y tridimensionales, con efectos especiales.

Wintermute – [116] - Herramienta orientada al desarrollo de aventuras gráficas en 2D. También admite el denominado 2.5D, en el cual los personajes son en 3D y los fondos en 2D, como por ejemplo Klonoa. Algunas de las ventajas que ofrece son: lenguaje de scripting, soporte de personajes 3D, parallax scrolling ó herramientas de desarrollo específicas, como por ejemplo para los escenarios (SceneEdit) ó los personajes (SpriteEdit). Además, tiene versión para realizar juegos para iOS.

Ilustración 48 - Wintermute

En la siguiente tabla se estudia la comparación entre distintas herramientas:

	XNA	Adventure Game Studio	Game Maker	M.U.G.E.N.
Soporte gráficos en 2D	Si	Si	Si	Si
Soporte gráficos en 3D	Si	Si	Si	No
Multiplataforma	Windows/ Xbox/Windows Phone 7	PC, Mac, Linux	Windows	Windows, Linux
Abstracción de entrada	Si	Si	Si	No
Sistema de audio	Si	Si	Si	Si
Programación	C#	Scripting	Scripting	Scripting
Orientado a género de videojuegos	Todos	Aventuras gráficas	Aventuras	Lucha

Tabla 1 - Comparativa alternativas XNA

Se ha elegido el framework XNA por su gran potencial, por la facilidad de uso que da la capa de abstracción y por la gran comunidad de desarrolladores que existe en la red.

Dados, también, los objetivos del proyecto, era mucho más valorable el usar XNA que las otras herramientas porque se buscaba un desarrollo más complejo con la Inteligencia Artificial, y las demás herramientas se quedaban bastante cortas sobre todo por su programación orientada al scripting. Por eso se comentó anteriormente que XNA es el más potente de los sistemas estudiados, dado los recursos y funcionalidades que ofrece su programación .NET.

Capítulo 4

Desarrollo

4. Desarrollo

4.1 Análisis

En este apartado se expondrán el diagrama de casos de uso, y la descripción textual de cada caso de uso y los requisitos de usuario, funcionales y no funcionales.

4.1.1. Casos de uso

A continuación se muestra el diagrama de casos de uso del sistema:

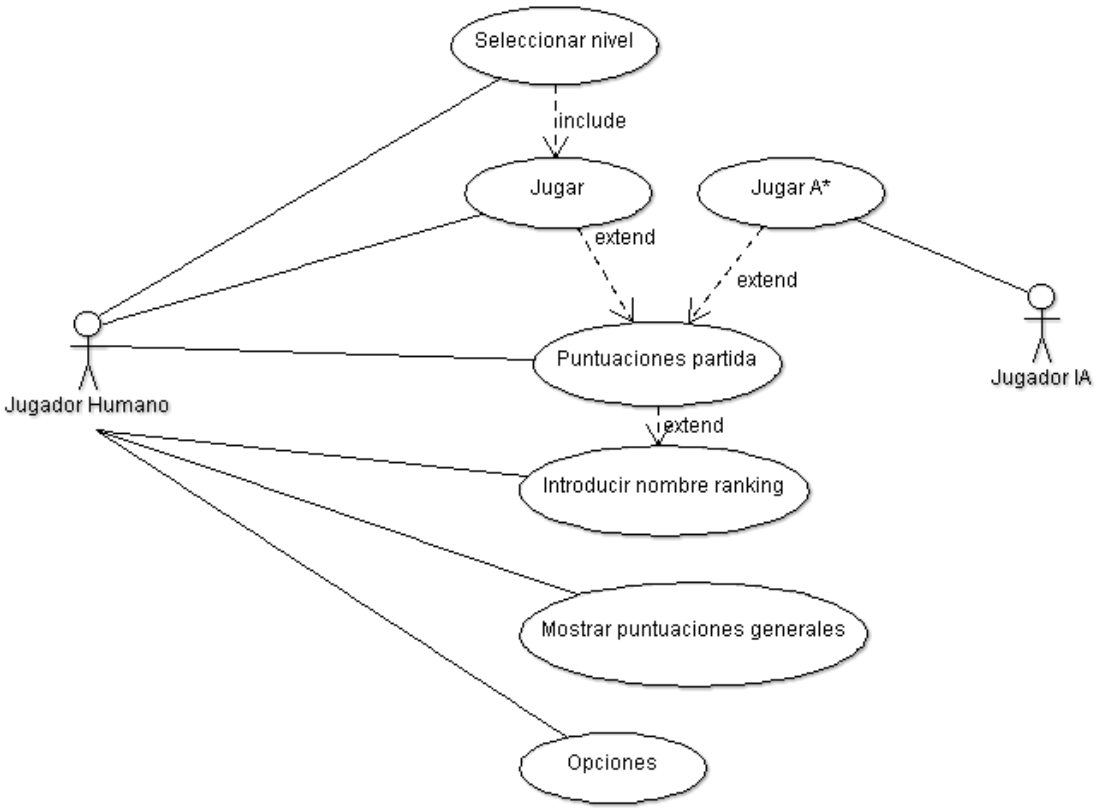


Ilustración 49 - Casos de uso

A partir de aquí se muestran la descripción de cada caso de uso:

Identificador	CU-01
Nombre	Jugar
Actores	Jugador humano
Descripción	El personaje del jugador tiene el objetivo de conseguir la máxima puntuación posible en función de las monedas recogidas y el tiempo restantes que le sobre completando el nivel.
Precondiciones	Haber seleccionado la opción Jugar en el Menú Principal o un nivel dentro del menú de selección de niveles
Postcondiciones	Partida finalizada y mostrar pantalla sus puntuaciones.

Tabla 2 - Caso de uso Jugar

Identificador	CU-02
Nombre	Jugar A*
Actores	Jugador IA
Descripción	El personaje del agente IA calcula el camino óptimo para maximizar la puntuación mediante un algoritmo de búsqueda A* y ejecuta las acciones pertinentes. El juego se desarrolla en tiempo real con el del jugador humano.
Precondiciones	Haber elegido la opción *Jugar* en el Menú Principal o un nivel dentro de la selección de niveles.
Postcondiciones	Partida finalizada y mostrar pantalla sus puntuaciones.

Tabla 3 - Caso de uso: Jugar A*

Identificador	CU-03
Nombre	Seleccionar nivel
Actores	Jugador humano
Descripción	Se muestra una lista de los niveles disponibles; y se carga el nivel que se seleccione para jugarlo.
Precondiciones	Haber elegido la opción *Seleccionar nivel* en el Menú Principal.
Postcondiciones	Nueva partida con el nivel seleccionado.

Tabla 4 - Caso de uso: Seleccionar nivel

Identificador	CU-04
Nombre	Puntuaciones partida
Actores	Jugador humano y jugador IA
Descripción	Se muestran las puntuaciones que han obtenido ambos personajes en la partida.
Precondiciones	Haber finalizado una partida.
Postcondiciones	Resultados por pantalla y menú de selección entre ver puntuaciones generales del nivel; o salir al menú principal.

Tabla 5 - Caso de uso: Puntuaciones partida

Identificador	CU-05
Nombre	Introducir nombre ránking
Actores	Jugador humano
Descripción	Si la puntuación que ha hecho el jugador entra dentro de las mejores del ránking. El jugador introduce su nombre y se guarda su puntuación en la posición correspondiente.
Precondiciones	Haber seleccionado la opción de *Ver puntuaciones generales* en el menú de *puntuaciones partida.*
Postcondiciones	Resultados de puntuaciones guardados en fichero (si procede) y rótulo de pulsar Enter para volver al menú principal.

Tabla 6 - Caso de uso: Introducir nombre ránking

Identificador	CU-06
Nombre	Mostrar puntuaciones generales
Actores	Jugador humano
Descripción	Se muestran las 5 máximas puntuaciones y el jugador que las consiguió del nivel seleccionado.
Precondiciones	Haber seleccionado la opción *Mostrar puntuaciones* en el Menú Principal.
Postcondiciones	Mostrar pantalla de puntuaciones.

Tabla 7 - Caso de uso: Mostrar puntuaciones generales

Identificador	CU-07
Nombre	Opciones
Actores	Jugador humano
Descripción	Cambiar las opciones del juego.
Precondiciones	Haber seleccionado *Opciones* en el Menú Principal.
Postcondiciones	Mostrar pantalla de Opciones.

Tabla 8 - Caso de uso: Opciones

4.1.2. Requisitos

A continuación se definen los requisitos de usuario:

4.1.2.1. Requisitos de usuario

Identificador	RU01
Nombre	Ejecutable
Descripción	El juego se lanzará a través de un ejecutable.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 9 - Requisito usuario 1

Identificador	RU02
Nombre	Jugar
Descripción	Se iniciará una nueva partida en el nivel por defecto (nivel1). El jugador humano y el de Inteligencia Artificial coinciden en el mismo escenario y tiempo. El objetivo para ambos es obtener la máxima puntuación con las monedas y el tiempo restante.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 10 - Requisito usuario 2

Identificador	RU03
Nombre	Selección de nivel
Descripción	Se muestran en pantalla todos los niveles disponibles para ser jugados. Al seleccionar uno de éstos, se carga dicho nivel y comienza la partida.
Origen	Seleccionar nivel
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 11 - Requisito usuario 3

Identificador	RU04
Nombre	Mostrar puntuaciones por nivel
Descripción	Se muestran en pantalla todos los niveles disponibles para consultar su puntuación.

Origen	Mostrar puntuaciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 12 - Requisito usuario 4

Identificador	RU05
Nombre	Mostrar puntuaciones partida
Descripción	Al finalizar una partida, se muestra la puntuación obtenida tanto el jugador humano como el jugador IA; así como el ganador de la partida.
Origen	Puntuaciones partida
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 13 - Requisito usuario 5

Identificador	RU06
Nombre	Introducir nombre ranking
Descripción	Tras comprobar la puntuación de la partida, pulsando en la opción correspondiente del menú se muestran las puntuaciones generales en ese nivel. Si se ha conseguido una puntuación que entra en el ranking, se introduce el nombre del jugador.
Origen	Introducir nombre ranking
Verificable	Sí
Claro	Sí
Prioridad	Alta

Necesidad	Esencial
Estabilidad	Alta

Tabla 14 - Requisito usuario 6

Identificador	RU07
Nombre	Jugar A*
Descripción	Juego del personaje del ordenador. Utilizando técnicas de Inteligencia Artificial (algoritmo A*), calcula el camino óptimo para maximizar la puntuación y lo ejecuta.
Origen	Jugar A*
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 15 - Requisito usuario 7

Identificador	RU08
Nombre	Opciones
Descripción	Mostrar opciones: • Activar/desactivar música • Configurar volumen
Origen	Opciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 16 - Requisito usuario 8

Identificador	RU09
Nombre	Control personaje
Descripción	El personaje se manejará a través del teclado. Podrá realizar las acciones de desplazamiento hacia la izquierda y hacia la derecha. También podrá saltar, y combinar este movimiento con los anteriores mencionados.
Origen	Jugar y Jugar A*
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 17 - Requisito usuario 9

Identificador	RU10
Nombre	Pausar partida
Descripción	Mientras se está jugando una partida, se puede pausar pulsando la tecla Escape. La partida queda "congelada" y el tiempo queda parado también. Se muestra un menú emergente con las posibilidades de volver a la partida en curso o al menú principal.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 18 - Requisito usuario 10

Identificador	RU11
Nombre	Guardar puntuación
Descripción	Si se consigue una puntuación que entra en el ránking de mayores puntuaciones, se guarda dicha puntuación, junto al nombre del usuario.
Origen	Introducir nombre ranking
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 19 - Requisito usuario 11

Identificador	RU12
Nombre	Confirmación salir
Descripción	Se pide la confirmación del usuario de que realmente se quiere volver al menú principal, para evitar que se pulse por equivocación y se pierda la partida.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 20 - Requisito usuario 12

Identificador	RU13
Nombre	Recoger monedas
Descripción	A lo largo del nivel hay monedas de distintos valores (500, 250 y 100) las cuales pueden ser recogidas por el personaje para aumentar su puntuación.

Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 21 - Requisito usuario 13

Identificador	RU14
Nombre	Puntuación tiempo
Descripción	Por cada segundo que sobre al jugador, se sumarán 50 puntos.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 22 - Requisito usuario 14

Identificador	RU15
Nombre	Enemigos
Descripción	Existen enemigos en cada nivel que si tocan al personaje le hacen soltar las monedas recogidas.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 23 – Requisito usuario 15

Identificador	RU16
Nombre	Jugador muerto.
Descripción	Si el jugador humano se queda sin monedas, muere. Por tanto, finaliza la partida
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 24 - Requisito usuario 16

4.1.2.3 Requisitos funcionales

A continuación se definen los requisitos funcionales:

Identificador	RF1
Nombre	Tecla selección menú
Descripción	La tecla que de selección de menú es el ENTER.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 25 - Requisito funcional 1

Identificador	RF2
Nombre	Tecla pausa.
Descripción	La tecla para pausar el juego es ESCAPE.
Origen	Pausar partida
Verificable	Sí

Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 26 - Requisito funcional 2

Identificador	RF3
Nombre	Tecla movimiento horizontal personaje.
Descripción	Para mover el personaje hacia la izquierda se pulsa flecha izquierda del teclado. Para su movimiento hacia la derecha, se pulsa la flecha derecha del teclado.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 27 - Requisito funcional 3

Identificador	RF4
Nombre	Salto personaje
Descripción	Para que el personaje salte, se pulsa la BARRA ESPACIADORA. Esta acción puede combinarse con el movimiento horizontal (RFXX)
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 28 - Requisito funcional 4

Identificador	RF5
Nombre	Volumen música
Descripción	En las opciones, se puede elegir un volumen entre 1 y 10.
Origen	Opciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 29 - Requisito funcional 5

Identificador	RF6
Nombre	Activar música
Descripción	Se puede activar o desactivar (Música: Si / No) en el menú opciones.
Origen	Opciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 30 - Requisito funcional 6

Identificador	RF7
Nombre	Caracteres nombre de jugador
Descripción	El nombre que introduce el jugador si consigue entrar en el ránking de puntuaciones puede contener caracteres de la A a la Z ([A-Z]), tanto en mayúsculas como en minúsculas.
Origen	Introducir nombre ránking
Verificable	Sí

Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 31 - Requisito funcional 7

Identificador	RF8
Nombre	Longitud nombre de jugador
Descripción	El nombre que introduce el jugador si consigue entrar en el ránking de puntuaciones puede tener una longitud máxima de 20 caracteres.
Origen	Introducir nombre ránking
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 32 - Requisito funcional 8

Identificador	RF9
Nombre	Número registros ránking
Descripción	El máximo número de puntuaciones que se guardarán en el ránking por cada nivel es de 5.
Origen	Guardar puntuaciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 33 - Requisito funcional 9

Identificador	RF 10
Nombre	Lanzar juego
Descripción	El juego se lanza con el ejecutable resultante de la compilación del proyecto.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 34 - Requisito funcional 10

Identificador	RF 11
Nombre	Salir del juego.
Descripción	Para salir del juego se pulsa en la opción *Salir* del menú principal. Pide confirmación de salida.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 35 - Requisito funcional 11

Identificador	RF 12
Nombre	Seleccionar niveles
Descripción	Se obtienen todos los niveles que haya en la carpeta /niveles. El nivel tiene que tener el siguiente formato de nombre: nivelX.txt, siendo un número comprendido entre 1 y 10.
Origen	Seleccionar nivel
Verificable	Sí

Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 36 - Requisito funcional 12

Identificador	RF 13
Nombre	Máximo número niveles
Descripción	El máximo número de niveles es 10.
Origen	Seleccionar nivel
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 37 - Requisito funcional 13

Identificador	RF 14
Nombre	Recoger monedas
Descripción	A lo largo del nivel se recogen monedas de distinto tipo las cuales tienen los siguiente valores en puntuación: • Moneda roja = 1000 puntos • Moneda verde: = 500 puntos • Moneda amarilla: = 250 puntos
Origen	Recoger monedas
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 38 - Requisito funcional 14

Identificador	RF 15
Nombre	Mostrar información monedas recogidas
Descripción	Durante la partida se muestran las monedas recogidas (de cada color) en ese momento por el personaje, tanto el humano como el del ordenador.
Origen	Recoger monedas
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 39 - Requisito funcional 15

Identificador	RF 16
Nombre	Soltar monedas
Descripción	Cuando un enemigo toca al personaje del jugador, éste suelta las monedas. Se tiran, como máximo, 12 monedas, siguiendo el siguiente criterio: • 6 amarillas. • 4 verdes. • 2 rojas. Si no se tuviesen suficientes de algún tipo, se va sumando el número que falta al siguiente tipo de moneda siguiendo la jerarquía de valores en puntuación.
Origen	Enemigos
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 40 - Requisito funcional 16

Identificador	RF 17
Nombre	Movimiento enemigos
Descripción	Los enemigos se mueven automáticamente horizontalmente hasta que encuentran un obstáculo en su camino. En ese momento, cambian el sentido de su movimiento, y así indefinidamente.
Origen	Enemigos
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 41 - Requisito funcional 17

Identificador	RF 18
Nombre	Vida enemigos
Descripción	Los enemigos no pueden morir, ya que los personajes de los jugadores no pueden dañar a los enemigos.
Origen	Enemigos
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 42 - Requisito funcional 18

Identificador	RF 19
Nombre	Mostrar tiempo restante
Descripción	Durante la partida se muestra el tiempo restante que queda para completar el nivel. Si dicho tiempo es menor de 10 segundos, éste cambia a color rojo para avisar de este hecho.

Origen	Enemigos
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 43 - Requisito funcional 19

Identificador	RF 20
Nombre	Duración niveles
Descripción	Cada nivel tiene un tiempo para ser completado. Éste valor se carga del fichero de nivel .txt en la primera línea.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 44 - Requisito funcional 20

Identificador	RF 21
Nombre	Volver al menú principal
Descripción	Cuando se pausa una partida, se puede seleccionar entre reanudar la partida o volver al menú principal. Si se selecciona ésta última, el sistema pide confirmación.
Origen	Pausar partida
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 45 - Requisito funcional 21

Identificador	RF 22
Nombre	Scroll horizontal
Descripción	Para niveles que ocupen más que la resolución de la pantalla, se implementa un scroll horizontal que seguirá siempre al jugador humano, mostrándole en el centro de la pantalla.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 46 - Requisito funcional 22

Identificador	RF 23
Nombre	Técnica Inteligencia Artificial jugador ordenador
Descripción	La técnica de Inteligencia Artificial por la que el jugador IA calculará sus movimientos es búsqueda A*.
Origen	Jugar IA
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 47 - Requisito funcional 23

4.1.2.3 Requisitos no funcionales

A continuación se definen los requisitos no funcionales:

Identificador	RNF 1
Nombre	Formato niveles

Descripción	Los niveles son en formato de texto plano .txt. Todas las filas tienen que tener la misma longitud de caracteres, es decir, que no puede haber filas con más caracteres que otros pues dará error.
Origen	Seleccionar nivel
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 48 - Requisito no funcional 1

Identificador	RNF 2
Nombre	Parallax scrolling
Descripción	Los fondos de los niveles tienen parallax scrolling.
Origen	Seleccionar nivel
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 49 - Requisito no funcional 2

Identificador	RNF 3
Nombre	Formato archivos gráficos.
Descripción	Los gráficos son formato .jpg o .png.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 50 - Requisito no funcional 3

Identificador	RNF 4
Nombre	Formato ficheros sonidos.
Descripción	Los sonidos son formato .wav.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 51 - Requisito no funcional 4

Identificador	RNF 5
Nombre	Resolución pantalla
Descripción	La resolución de la pantalla del juego es 800 x 480 píxeles.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 52 - Requisito no funcional 5

Identificador	RNF 6
Nombre	Control (input)
Descripción	El control de la aplicación se hará única y exclusivamente por entrada de teclado.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 53 - Requisito no funcional 6

Identificador	RNF 7
Nombre	Sistemas Operativos compatibles
Descripción	El juego se puede ejecutar en los sistemas Windows XP, Windows Vista y Windows 7.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 54 - Requisito no funcional 7

Identificador	RNF 8
Nombre	Formato ficheros puntuaciones.
Descripción	El formato de los ficheros de puntuación es .xml.
Origen	Introducir puntuaciones.
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 55 - Requisito no funcional 8

Identificador	RNF 9
Nombre	Requisitos hardware PC
Descripción	Los requisitos mínimos de hardware que ha de tener un PC para poder ejecutar correctamente la aplicación son: • Procesador 1.5 Ghz • RAM: 1 GB Disco Duro: 20 MB
Origen	Ejecutable
Verificable	Sí

Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 56 - Requisito no funcional 9

Identificador	RNF 10
Nombre	Requisitos software
Descripción	Los requisitos mínimos de hardware que ha de tener un PC para poder ejecutar correctamente la aplicación son: • Sistema operativo compatible (RNF XX) • XNA Framework Redistributable 4.0 http://www.microsoft.com/download/en/details.aspx?id=20914 .
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 57 - Requisito no funcional 10

Identificador	RNF 11
Nombre	Tamaño
Descripción	El máximo tamaño que puede tener el juego es de 20 Megabytes.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 58 - Requisito no funcional 11

Identificador	RNF 12
Nombre	Instalador
Descripción	El juego tendrá un instalador.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 59 - Requisito no funcional 12

Identificador	RNF 13
Nombre	Tratamiento excepciones
Descripción	El juego contendrá tratamiento de excepciones.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 60 - Requisito no funcional 13

Identificador	RNF 14
Nombre	Tiempo de carga puntuaciones
Descripción	El tiempo de carga de las puntuaciones será menor de 1 segundo.
Origen	Mostrar puntuaciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 61 - Requisito no funcional 14

Identificador	RNF 15
Nombre	Entorno de desarrollo
Descripción	El entorno de desarrollo es Microsoft Visual Studio 2010 Professional.
Origen	Mostrar puntuaciones
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 62 - Requisito no funcional 15

Identificador	RNF 16
Nombre	Lenguaje de programación
Descripción	El lenguaje de programación es C#.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 63 - Requisito no funcional 16

Identificador	RNF 17
Nombre	Versión XNA
Descripción	La versión de XNA es la 4.0.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 64 - Requisito no funcional 17

Identificador	RNF 18
Nombre	Interfaz menú
Descripción	La interfaz del menú es con elementos gráficos y con visualización clara del elemento seleccionado. Para esta visualización clara el elemento seleccionado tiene un color distinto y un tamaño variable en comparación a los demás elementos.
Origen	
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 65 - Requisito no funcional 18

Identificador	RNF 19
Nombre	Idioma
Descripción	El idioma del juego es el castellano.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 66 - Requisito no funcional 19

Identificador	RNF 20
Nombre	Pruebas
Descripción	Se realizarán pruebas para verificar el juego.
Origen	Ejecutable
Verificable	Sí
Claro	Sí
Prioridad	Alta

Necesidad	Esencial
Estabilidad	Alta

Tabla 67 - Requisito no funcional 20

Identificador	RNF 21
Nombre	Manual del juego
Descripción	Realización de un manual del juego con su explicación y capturas de las pantallas.
Origen	Jugar
Verificable	Sí
Claro	Sí
Prioridad	Alta
Necesidad	Esencial
Estabilidad	Alta

Tabla 68 - Requisito no funcional 21

4.1.3 Diagramas de flujo

En esta sección se muestran los diagramas de diseño realizados en función de los requisitos.

4.1.3.1. Diagrama de flujo general

Diagrama de flujo del uso general de la aplicación.

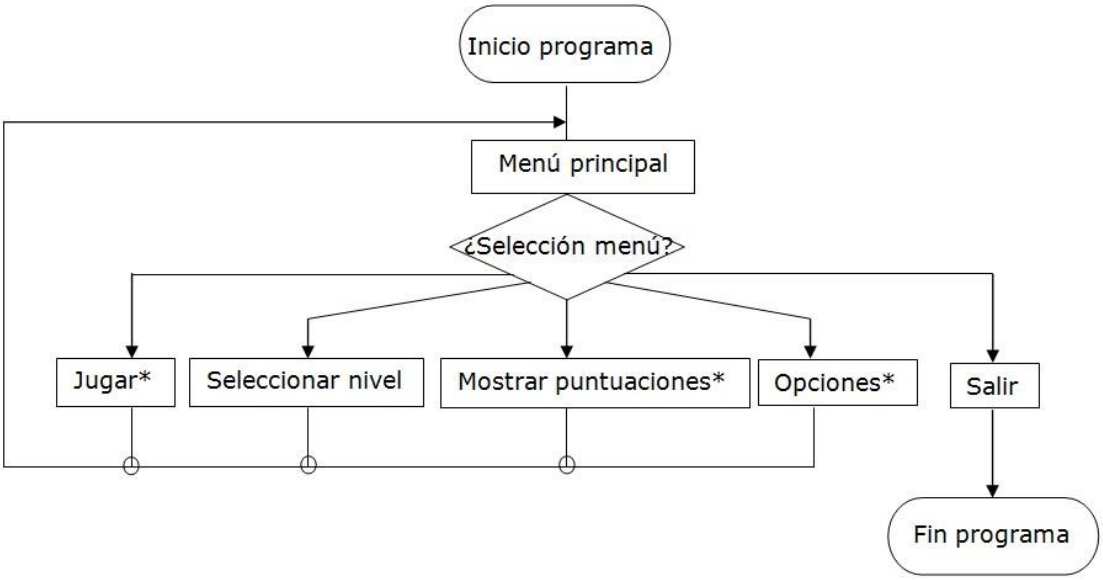


Ilustración 50 - Diagrama de flujo general

Este diagrama general se puede apreciar su posterior aplicación al menú principal del juego:

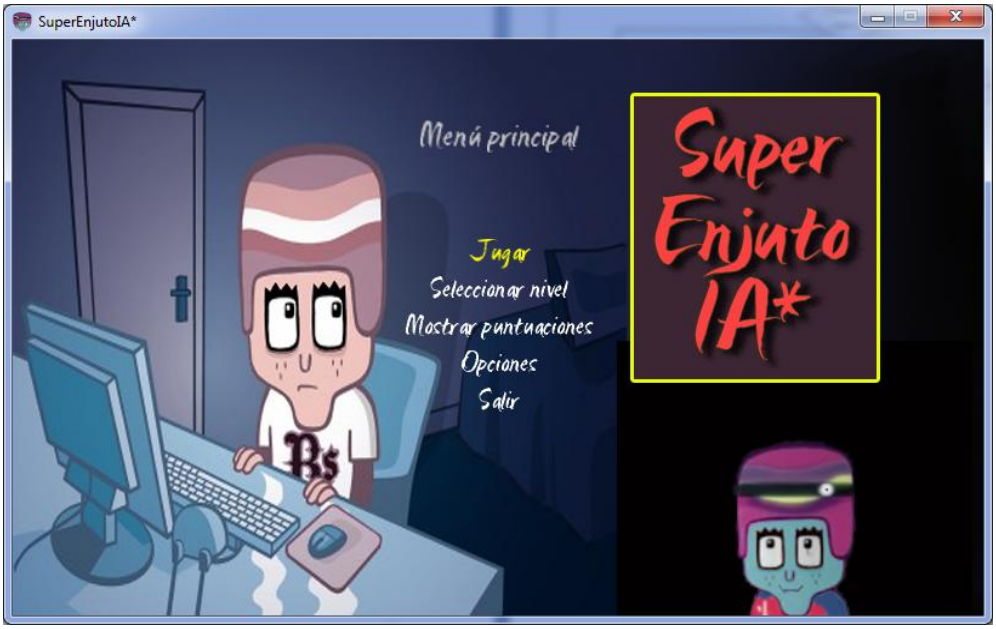


Ilustración 51 - Menú principal

4.1.3.2 Diagrama de flujo jugar

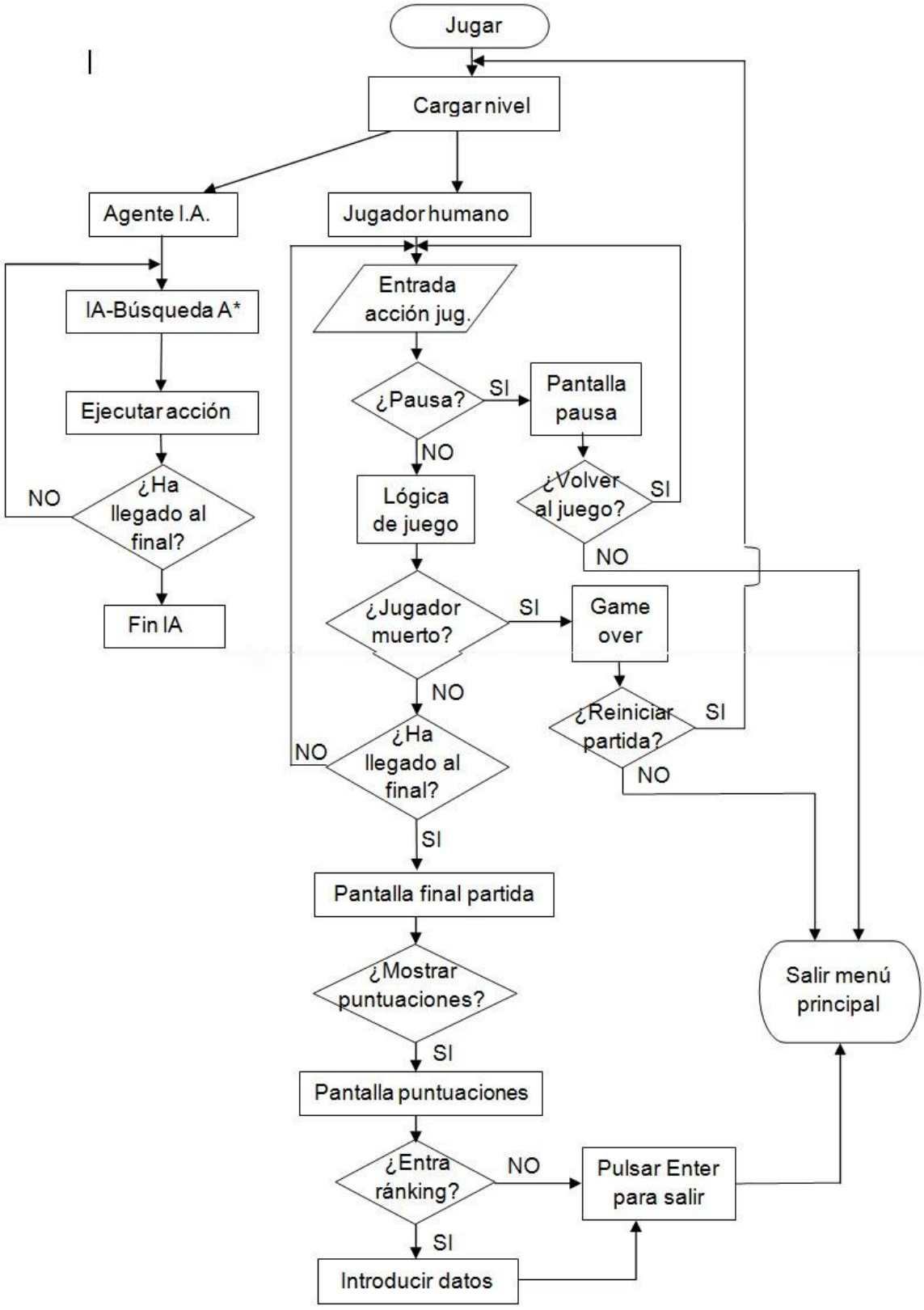


Ilustración 52 - Diagrama de flujo jugar

4.1.3.3 Diagrama de flujo - Seleccionar nivel

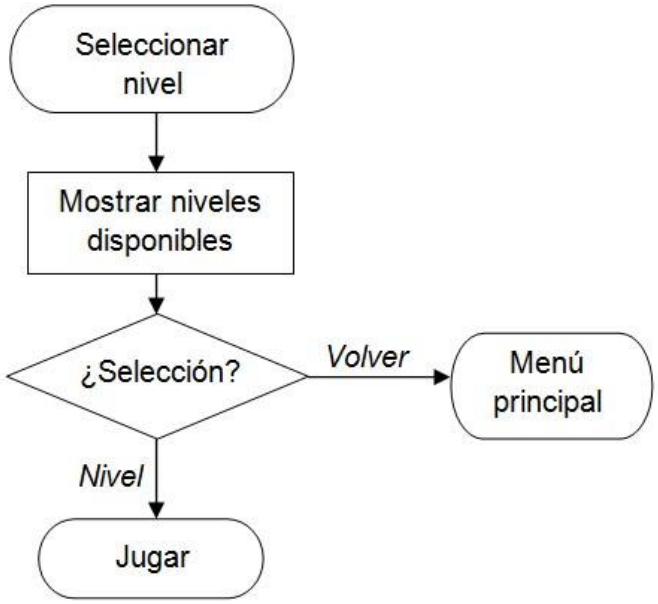


Ilustración 53 - Diagrama de flujo seleccionar nivel

4.1.3.4. Diagrama de flujo - Mostrar puntuaciones

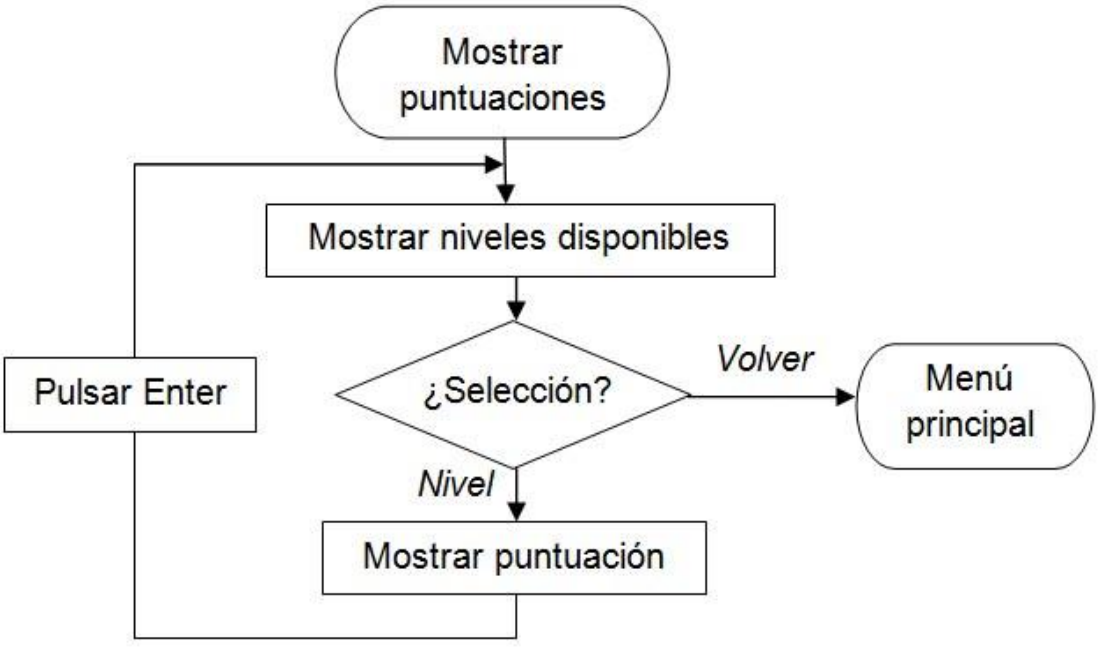


Ilustración 54 - Diagrama de flujo mostrar puntuaciones

4.1.3.5 Diagrama de flujo - Opciones

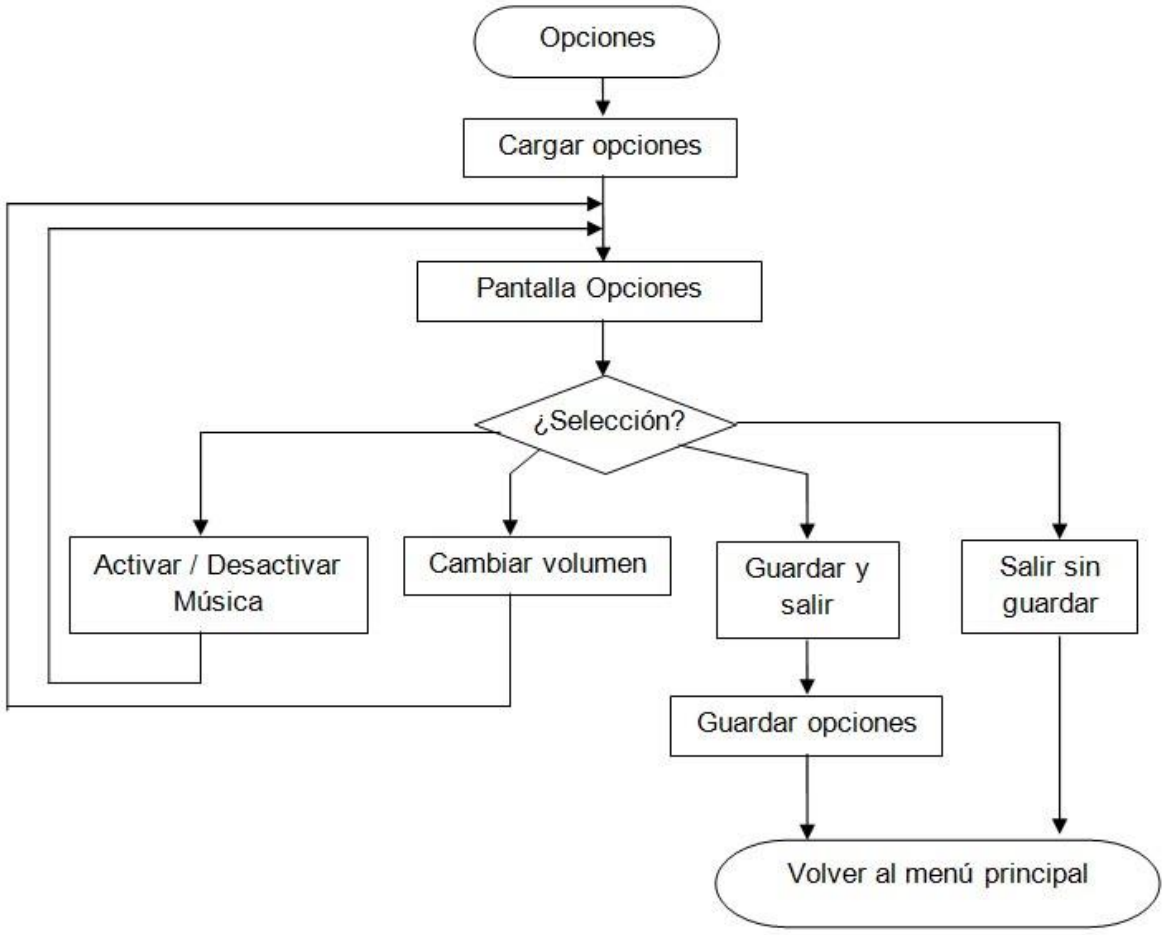


Ilustración 55 - Diagrama de flujo opciones

4 Diagrama de actividad

A continuación se muestra el diagrama de actividad general de la aplicación, en especial orientado a la gestión entre pantallas:

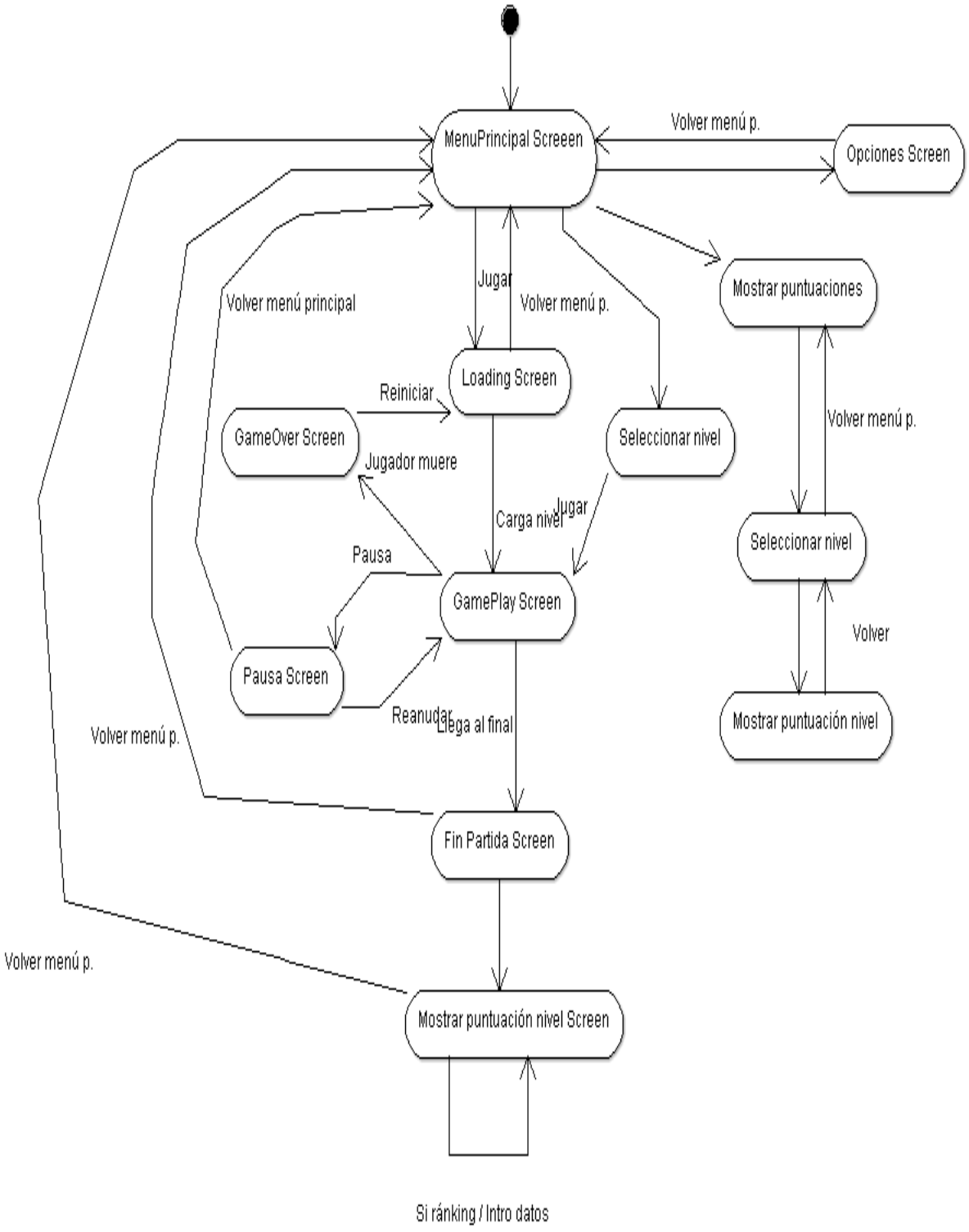


Ilustración 56 - Diagrama de actividad

4.2 Diseño conceptual

4.2.1. Arquitectura del sistema

Diagrama de la descomposición del juego en módulos

A continuación se muestra el diagrama de descomposición del juego en módulos, con sus interacciones entre ellos.

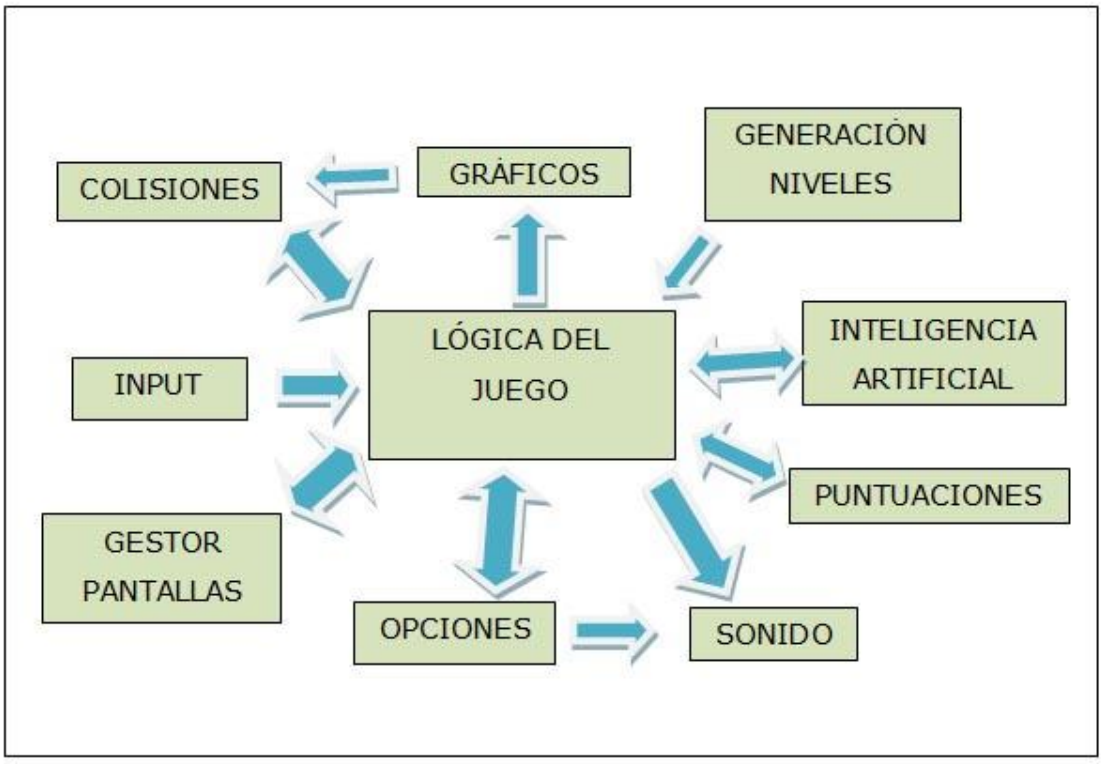


Ilustración 57 - Arquitectura del sistema por módulos

4.2.2. Descripción de los módulos

Una vez presentada la interacción entre módulos, se pasa a explicara cada módulo por separado.

4.2.2.1. Gestión de pantallas

Para la gestión entre pantallas, se ha utilizado de base el recurso *Game State Management*, el cual ofrece Microsoft para la comunidad de desarrolladores Xna Creators [117]. Es la versión 4.0, en la que se introduce también el soporte para Windows Phone 7 y se añaden nuevos funcionalidades y efectos de transición. El objetivo de esta librería es proporcionar una gestión eficiente entre las distintas pantallas que se requieran en el juego. En dicha referencia, se muestra un proyecto de ejemplo en el que se usa crean y eliminan pantallas, transición entre éstas, lógica, así como la gestión de menús y submenús con sus eventos. A raíz de ello, se han adaptado y añadido funcionalidades para adaptarlo a los objetivos de este proyecto.

En la siguiente figura se muestra el diseño de las clases del ScreenManager:

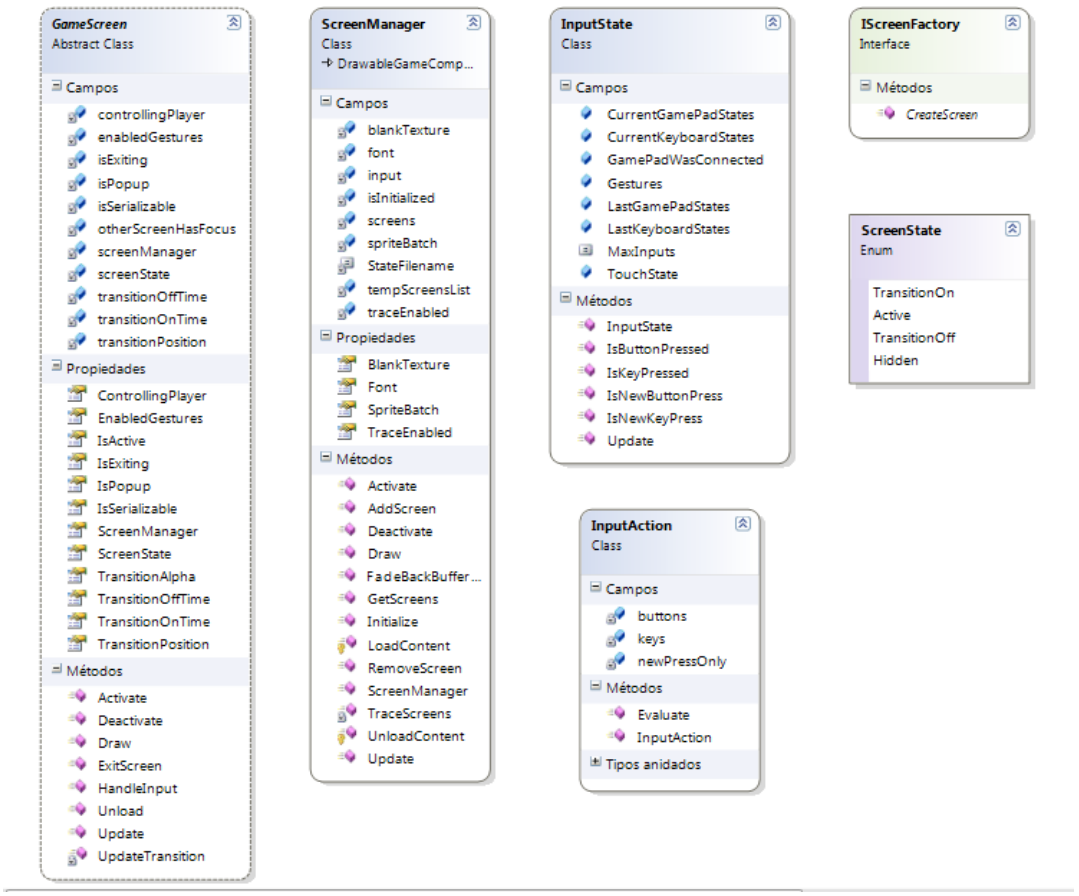


Ilustración 58 - Clases ScreenManager

La clase ScreenManager es el componente que lleva la base de la gestión. Va almacenando una pila de pantallas (clase GameScreen). Se encarga de coordinarlas: tanto sus transiciones, como la lógica de sus menús, así como de la entrada por parte del usuario. Para esta última labor, utiliza las clases InputState e InputAction.

Además del motor principal de la gestión entre pantallas, se proporcionan también ejemplos que modelan las clases que han de tener dichas pantallas. Todas ellas heredan de la clase anteriormente tratada GameScreen. Son las siguientes:

- *BackgroundScreen* – Pantalla de fondo de menú. En ella se cargan las imágenes que se mostrarán como fondo del menú.

- *GameplayScreen* – Pantalla en la que se desarrolla la acción del juego.

- *LoadingScreen* – Pantalla que se muestra mientras se están cargando los elementos y recursos del juego.

- *MainMenuScreen* – Pantalla que muestra el menú principal del juego.

- *MenuEntry* – Clase que representa a cada uno de los elementos que conforman un menú; con sus manejadores de eventos correspondientes.

- *MenuScreen* – Hereda de MainMenuScreen. Clase que representa un menú de opciones.

- *MessageBoxScreen* – Pantalla que muestra una menú emergente de forma modal. Se utiliza por ejemplo para pedir la confirmación de usuario al salir de una partida, con la finalidad de evitar salidas del juego por equivocación.

• *OptionsMenuScreen* – Hereda de MenuScreen. Pantalla de opciones en la que se puede cambiar la configuración de distintos elementos a través de los menús.

• *PauseMenuScreen* – Hereda de MenuScreen. Pantalla de pausa dentro del juego que muestra las opciones de menú correspondientes.

• *PlayerIndexEventArgs* – Clase para especificar qué usuario ha realizado la acción. En este proyecto en concreto no se utiliza, pero es útil para juegos con varios usuarios para saber qué jugador ha dado la entrada a la aplicación, por ejemplo al menú de pausa.

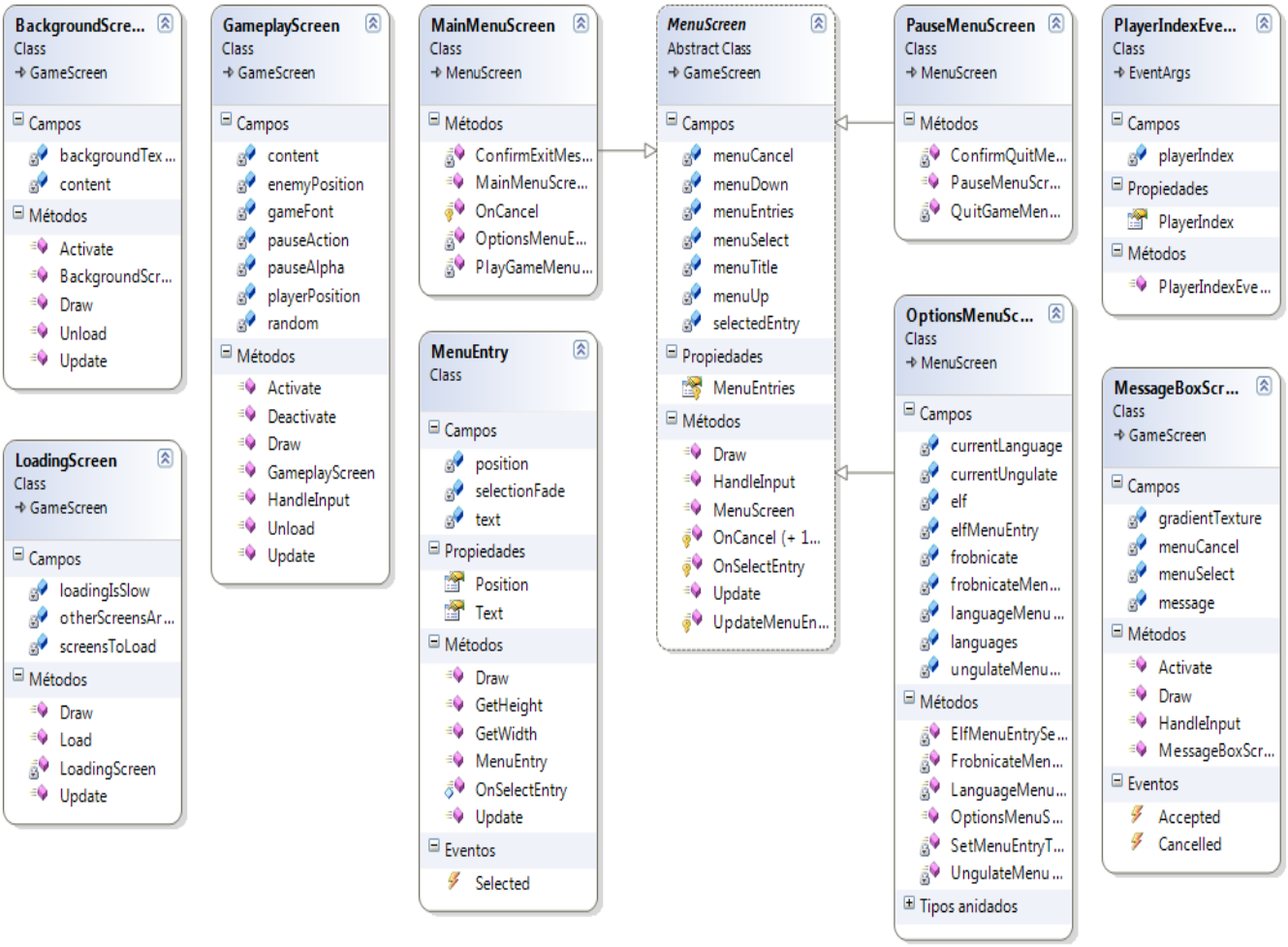


Ilustración 59 - Clases gestión de pantallas

Como se ha dicho anteriormente, éstas son las clases de las pantallas que proporcionaba el recurso del XnaCreator *GameStateManager*. Pero, para adaptarlo a los contenidos del juego, se han creado y modificado nuevas clases, utilizando las anteriores de base, las cuales se exponen a continuación.

- Pantallas de background:

  - *BackgroundFinScreen* – Pantalla de fondo para cuando se finaliza una partida.
  - *BackgroundGameOverScreen* – Pantalla de fondo cuando un jugador muere.
  - *BackgroundMenuPrincipalScreen* – Pantalla de fondo en el menú principal.

- Pantallas de menú:

  - *FinPartidaMenuScreen* – Menú que se muestra al completar una partida.

  - *GameOverMenuScreen* – Menú que se muestra cuando se acaba una partida porque el personaje muere o se acaba el tiempo.

  - *MainMenuScreen* – Menú principal.

  - *MenuScreen* – Clase de la que heredan los demás menús. Contiene los métodos para el posicionamiento y el dibujo de los elementos que componen el menú.

  - *MostrarPuntuacionesMenuScreen* – Menú para elegir el nivel del que se quiere cargar la puntuación.

- *OptionsMenuScreen* – Menú de opciones. Permite la activación y desactivación de la música, así como modificar el volumen.

- *PauseMenuScreen* – Menú de pausa que permite volver al juego o salir al menú principal.

- *SeleccionarNivelMenuScreen* – Menú para seleccionar el nivel que se quiere jugar.

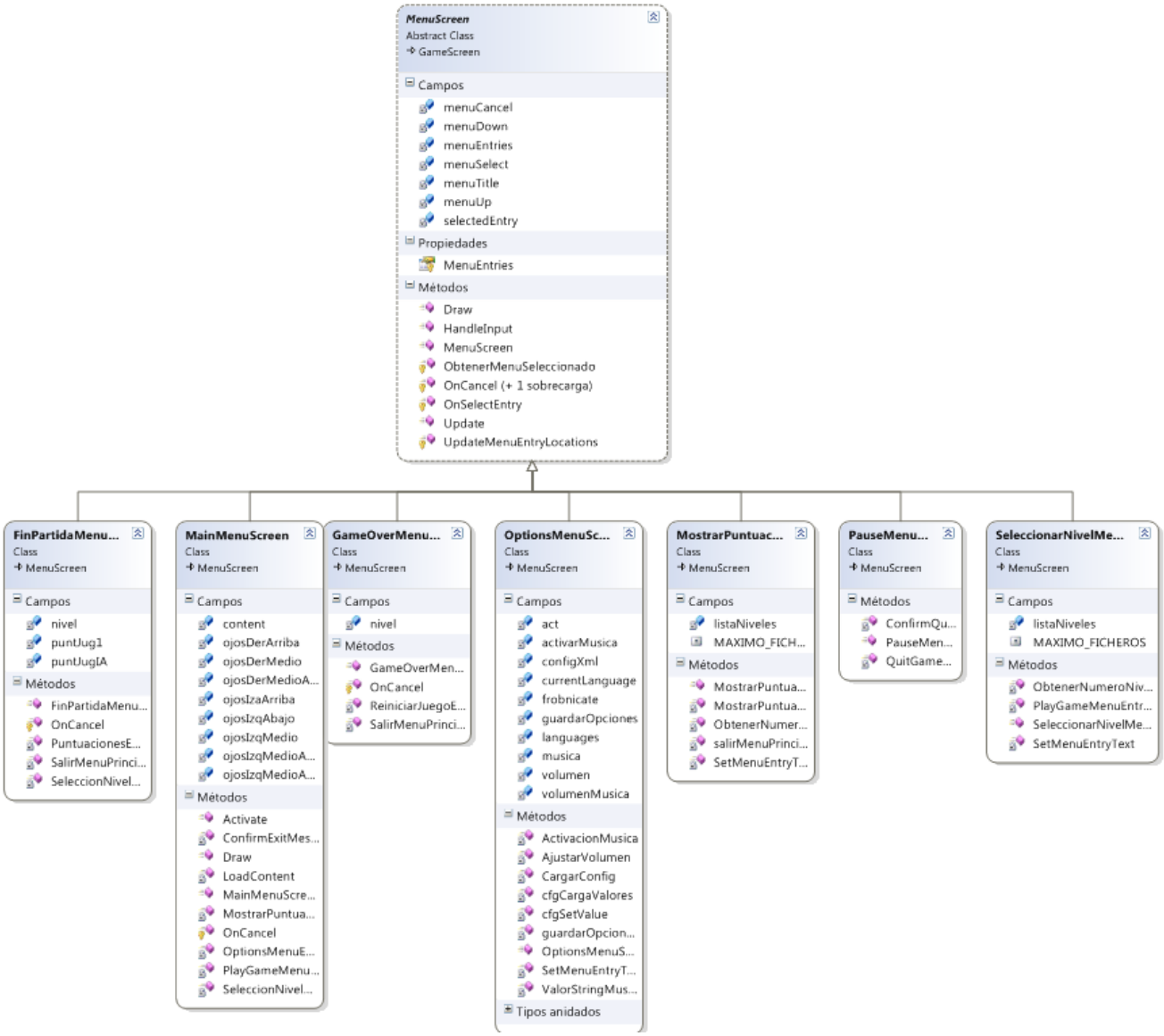


Ilustración 60 - Clases gestión pantallas 2

- Pantallas de juego:

  - *GamePlayScreen* – Pantalla en la que se carga y desarrolla el nivel del juego. En su LoadContent se llama método CargarNivel que genera una nueva instancia de nivel. Además, es esta clase la encargada de dibujar el contador de monedas y el tiempo restante.

  - *MostrarPuntuacionesPorNivel* – Pantalla que muestra las puntuaciones obtenidas tanto por el jugador humano como por el jugador del ordenador. Muestra, además, quien ha resultado vencedor.

  - *PuntuacionesScreen* – Pantalla en la que se muestran las puntuaciones almacenadas para el nivel en cuestión. A esta pantalla se le puede llamar por dos caminos: o desde el menú de mostrar puntuaciones, o al finalizar una partida y haber pasado ya las puntuaciones del nivel (MostrarPuntuacionesPorNivel). En este último caso, si el jugador ha hecho una puntuación que entra en el ránking, se captura el nombre del jugador y al pulsar Enter se guarda dicha puntuación.

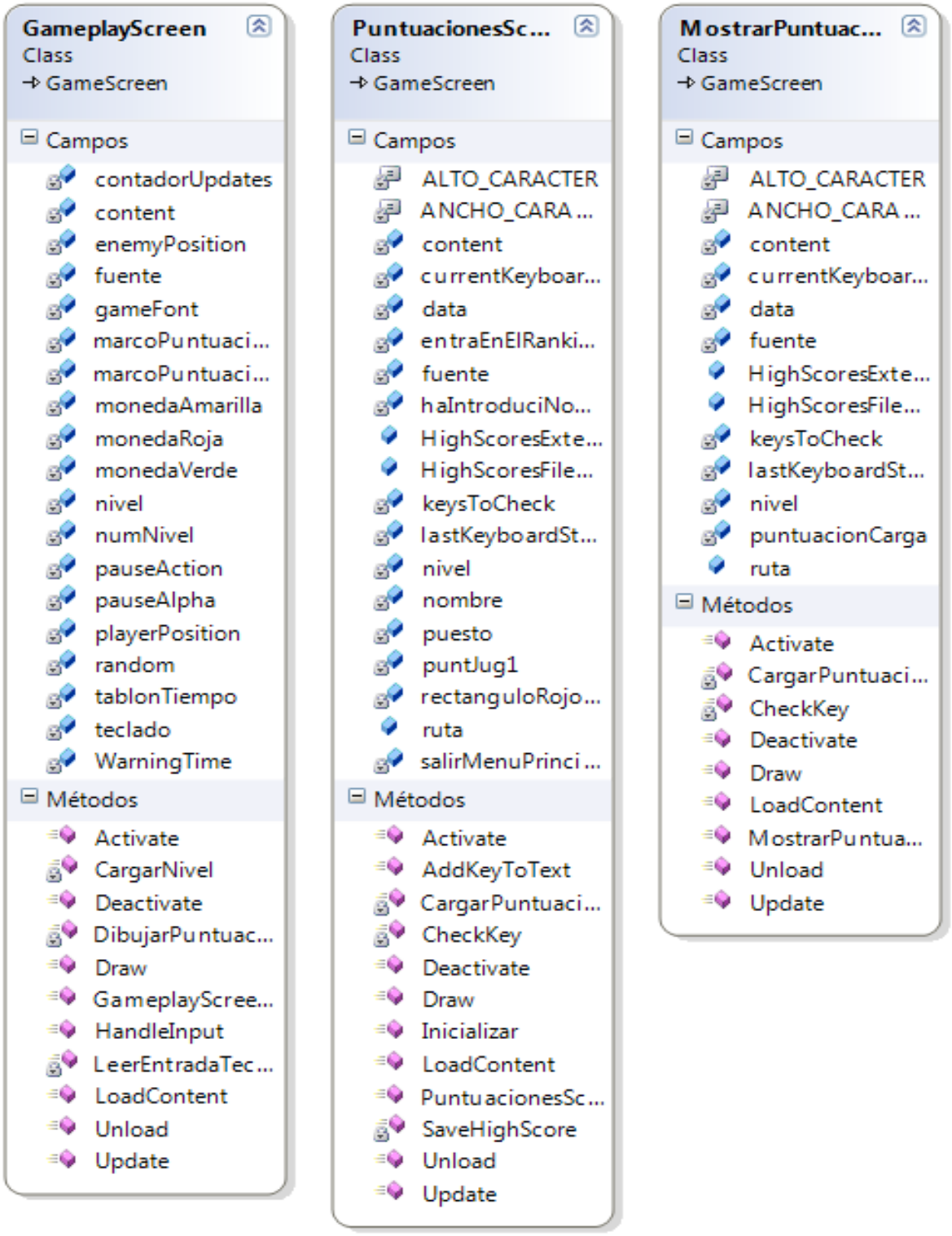


Ilustración 61 - Clases gestión pantallas 3

4.2.2.2 Opciones

El módulo de opciones se utiliza para configurar las opciones de la música. Esto incluye tanto la activación / desactivación de la misma, como configurar el su volumen. En este punto hay que aclarar que si la música está desactivada, da igual el volumen al que se configure. Del mismo modo, si la música está activada, y se selecciona un volumen de 0, no se reproducirá ningún sonido.

El fichero de opciones es un fichero en formato XML. Su nombre es App.config y está en la ruta de instalación del programa. El formato es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="musica" value="No" />
    <add key="volumen" value="10" />
  </appSettings>
</configuration>
```

En las claves *música* y *volumen* se guarda la configuración respectivamente. Para el tratamiento de las opciones se utiliza la clase OptionsMenuScreen. Sus métodos serán explicados en detalle en el apartado Implementación de este documento.

4.2.2.3. Puntuaciones

El módulo de puntuaciones se utiliza para guardas las máximas puntuaciones de cada nivel. Se guardan 5 registros de máxima puntuación por nivel. Sólo se guarda la puntuación del jugador

humano. Se almacena tanto la puntuación del jugador obtenida, como el nombre del usuario. La longitud máxima del nombre de usuario es de 20 caracteres. Los caracteres que se pueden utilizar van de la A a la Z (excluyendo la 'Ñ') tanto minúsculas como mayúsculas.

Los ficheros de puntuaciones se guardan en la carpeta Puntuaciones dentro del directorio de la aplicación. Dichos ficheros tienen el nombre puntNivelX.lst, siendo X el número correspondiente al nivel de la puntuación. La extensión .lst corresponde a que es un fichero XML. Su formato es el siguiente:

```
<?xml version="1.0"?>
<HighScoreData
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <PlayerName>
    <string>Vacio</string>
    <string>Vacio</string>
    <string>Vacio</string>
    <string>Vacio</string>
    <string>Vacio<string />
  </PlayerName>
  <Score>
    <int>0</int>
    <int>0</int>
    <int>0</int>
    <int>0</int>
    <int>0</int>
  </Score>
  <Count>5</Count>
</HighScoreData>
```

Los nombres de los jugadores están inicializados a *Vacio*, y las puntuaciones a *0*. En la sección *Implementación* se explica en detalle el funcionamiento de este módulo.

4.2.2.4. Sonido

Se ha utilizado el propio motor de sonido que ofrece Xna con su framework, en concreto el apartado de Audio (Microsoft.Xna.Framewor.Audio). Existe tanto música de fondo; como sonidos de acciones.

La música se ha obtenido de la referencia [118]. En ella, se puede descargar música libre para videojuegos. La canción utilizada en este proyecto es *Dancing,* y su autor es Webmaster24.

En cuanto a los sonidos de acciones, se han obtenido de la propia serie de Enjuto Mojamuto [5] . Se han utilizado dos sonidos, uno correspondiente a cuando el jugador muere; y otro reproducido cuando el jugador completa el nivel.

La clase que gestiona la reproducción de sonidos es Nivel.cs. Cuando queda poco tiempo, es decir, se entra en el WarningTime al quedar menos de 15 segundos, se acelera el volumen de la música para avisar también.

4.2.2.5 Gráficos

El módulo de gráficos puede clasificarse en varias categorías según la funcionalidad que realiza, por ejemplo pintar personajes, niveles, o rótulos. Sin embargo, para todas ellas, se han utilizado las funcionalidades que ofrece Xna para el tratamiento de gráficos.

(Microsoft.Xna.Framework.Graphics). El elemento básico para dibujar es el sprite, es decir, la textura a pintar por pantalla.

4.2.2.6 Parallax – Scrolling.

Son las imágenes que se dibujan al fondo del nivel y que se mueven horizontalmente para aportar un efecto de velocidad y paso del tiempo. Hay dos posibles escenarios de parallax – scrolling, uno que simboliza al día, y otro a la noche. Para el funcionamiento de este módulo se parte de la clase CapaParallax. Esta clase representa cada una de las capas que se dibujan al final. Cada una de éstas tiene su propiedad de velocidad de movimiento propia. El movimiento indica la velocidad a la que el sprite se va desplazando hacia la derecha. De esta manera, al tener cada capa una velocidad distinta, produce el efecto deseado.

4.2.2.7 Colisiones

El sistema de colisiones que se ha usado como base se ha obtenido del juego de ejemplo y tutoriales existente en la comunidad Xna Creators, llamada Platformer [119] Se han hecho modificaciones para adaptarlo al entorno del proyecto; sobre todo con el sistema de sprites.

En la parte de implementación se explicará en detalle cómo funciona el mecanismo detector de colisiones.

4.2.2.8 Generación niveles

Se cargan desde un fichero txt. Se ha hecho de esta manera para que sea más fácil la pre visualización de la morfología del nivel; sin necesidad de cargarlo en la aplicación para tener una visión general de cómo es. Además, hace más fácil al usuario poder hacer

sus propios niveles simplemente creando nuevos ficheros con el formato adecuado.

El diseño de los niveles debe seguir unas normas y requisitos básicos para su correcto funcionamiento:

- La primera línea del fichero indica el tiempo del nivel. Este número indica el máximo de tiempo que se dispone para superar el nivel. Una vez que la partida comienza, dicho tiempo irá corriendo hacia atrás. Si llegase a 0, la partida finalizaría. Indicar que cuando faltan 10 segundos se cambia el color del tiempo para avisar al jugador. El formato que ha de seguir esta línea es el siguiente: M:SS , siendo M los minutos, y SS los segundos. Hay que respetar este formato estrictamente, ya que de otro modo la aplicación fallará indicando el motivo.

- Las siguientes líneas corresponden a la generación propia del nivel. Todas las líneas tienen que tener la misma longitud. Si no fuese así, la aplicación falla avisando del error. En la sección implementación se detallará en profundidad el método de carga de niveles.

4.2.2.9 Input

La entrada de las acciones del usuario se hacen a través del periférico del teclado exclusivamente. Los controles son los siguientes:

- Desplazamiento hacia la derecha: Flecha derecha (→)

- Desplazamiento hacia la izquierda: Flecha izquierda (←)

- Salto: Barra espaciadora

Para el gestor de ventanas, se ha usado la clase destinada a tal efecto en su librería ActionInput. Sin embargo, para la partida se ha implementado una toma de entrada propia especificada para el juego, la cual será detallada en el apartado correspondiente de la implementación.

4.2.2.10 Lógica del juego

A parte de la ya comentada gestión de ventanas, la gran lógica del juego y sus partidas se lleva dentro en la clase Nivel. Ésta se crea a partir de la pantalla GamePlayScreen. La clase nivel es que lleva todo el peso, al contener en ella las demás clases, como Jugador, Enemigo y Monedas (además de sus propiedades).

En la siguiente figura se muestra el diagrama de secuencia para la comprensión de la lógica del juego. Se ha detallado a nivel general el método Update.

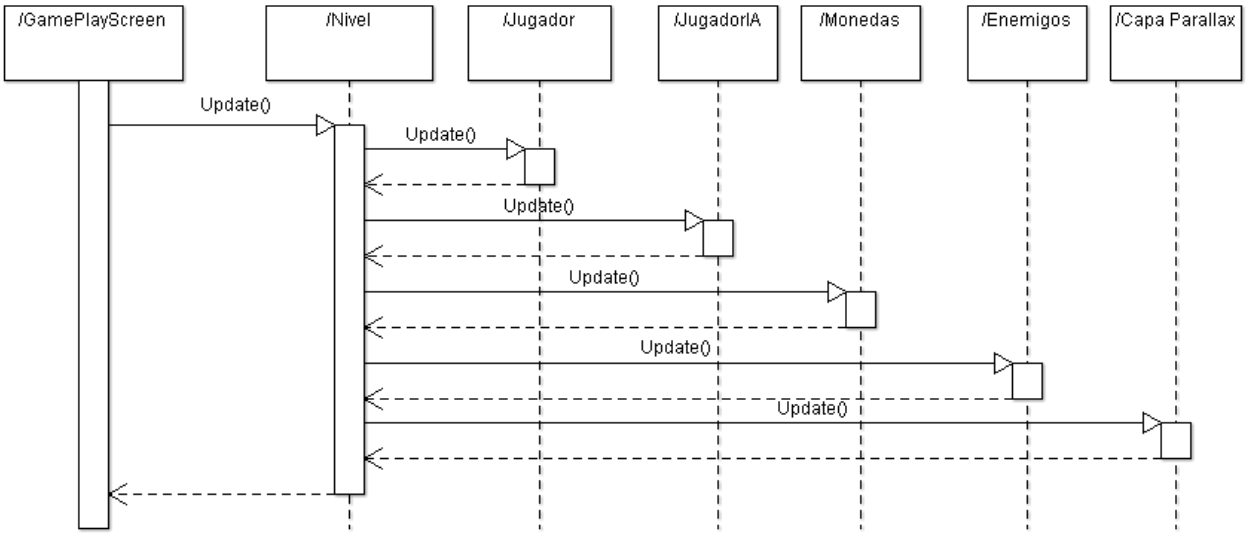


Ilustración 62 - Diagrama de secuencia lógica del juego

4.2.2.11 Inteligencia Artificial

La técnica de inteligencia artificial que se ha usado en este proyecto ha sido la búsqueda de caminos (pathfinding) con el algoritmo A *. A través de él, el personaje del ordenador obtiene las acciones a realizar. El algoritmo A* es muy potente, pero también requiere muchos recursos computacionales. Por eso, el diseño de este módulo tiene que estar orientado también hacia su eficiencia y eficacia, ya que se dispone de unos requisitos de tiempo y máquina específicos. Para ello, se aplicarán técnicas de optimización las cuales serán explicadas también en la sección de implementación.

4.2.3. Diagrama de clases

La primera aproximación que se va a realizar al diagrama de clases del sistema es a raíz de la arquitectura del sistema anteriormente descrita. A partir del esquema de los módulos del juego y su interrelación entre sí, se van a situar las clases del juego en función del módulo que las contiene. El módulo gestor de pantallas contiene más clases de las expuestas en el

diagrama, pero se ha diseñado esta figura así para un mejor entendimiento. Las clases contenidas en este módulo son todas las anteriormente descritas en el apartado *Gestión de pantallas.*

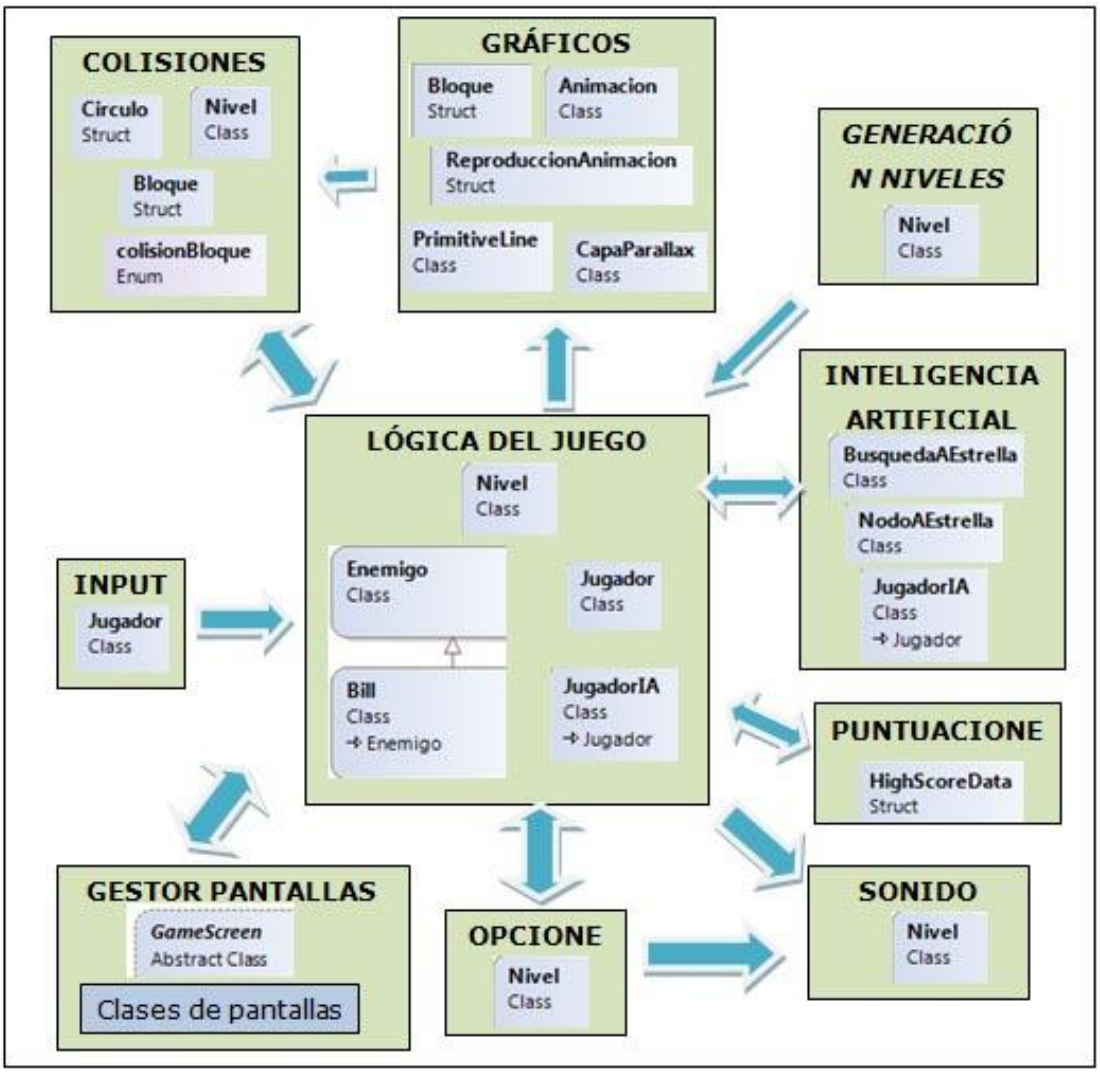
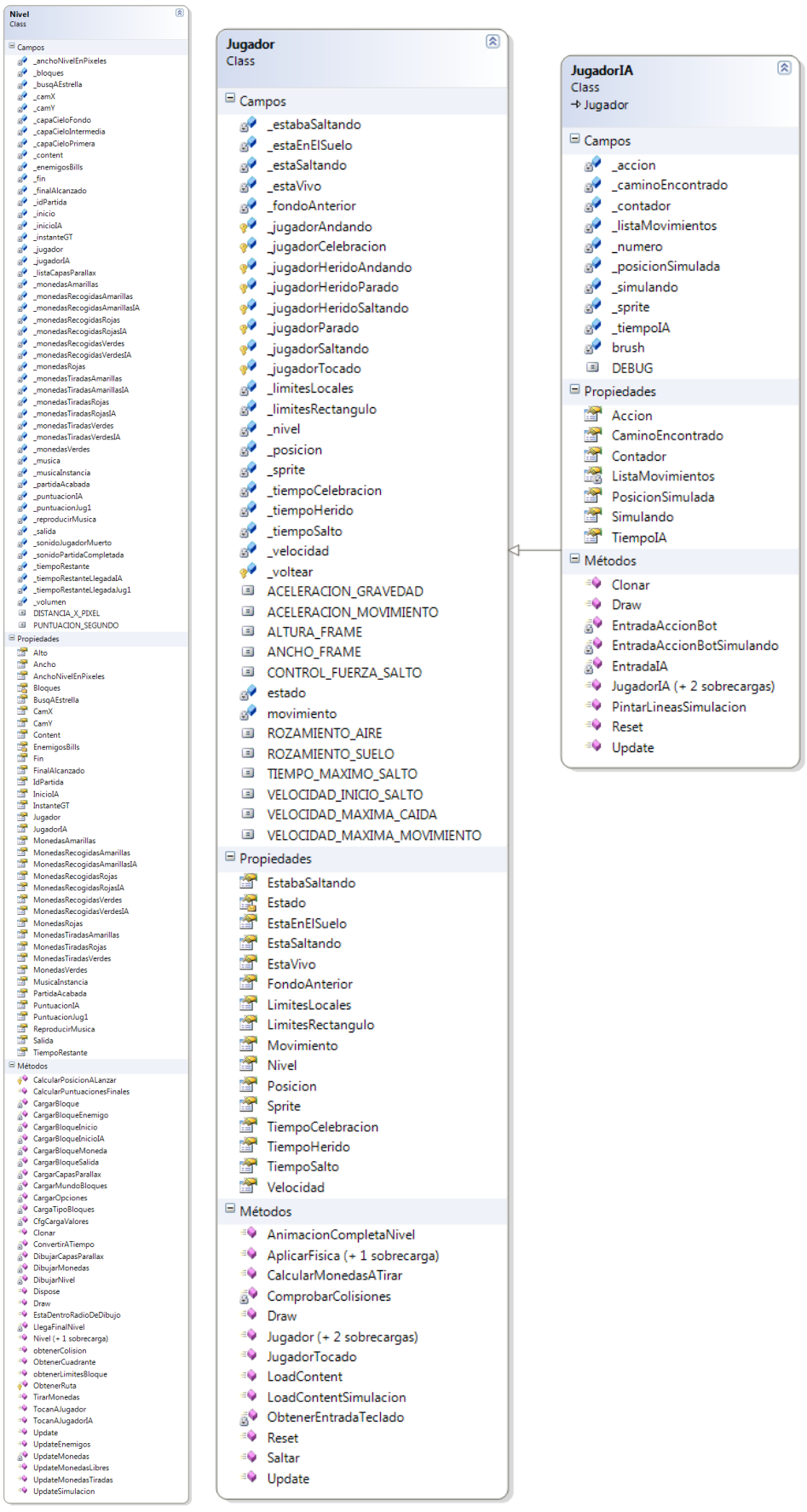


Ilustración 63 - Arquitectura de sistema con clases

A continuación se muestran todas las clases que intervienen en la aplicación (aparte de las de gestión entre pantallas, que ya ha sido explicada en puntos anteriores).



Ilustación 65 – Clases 1

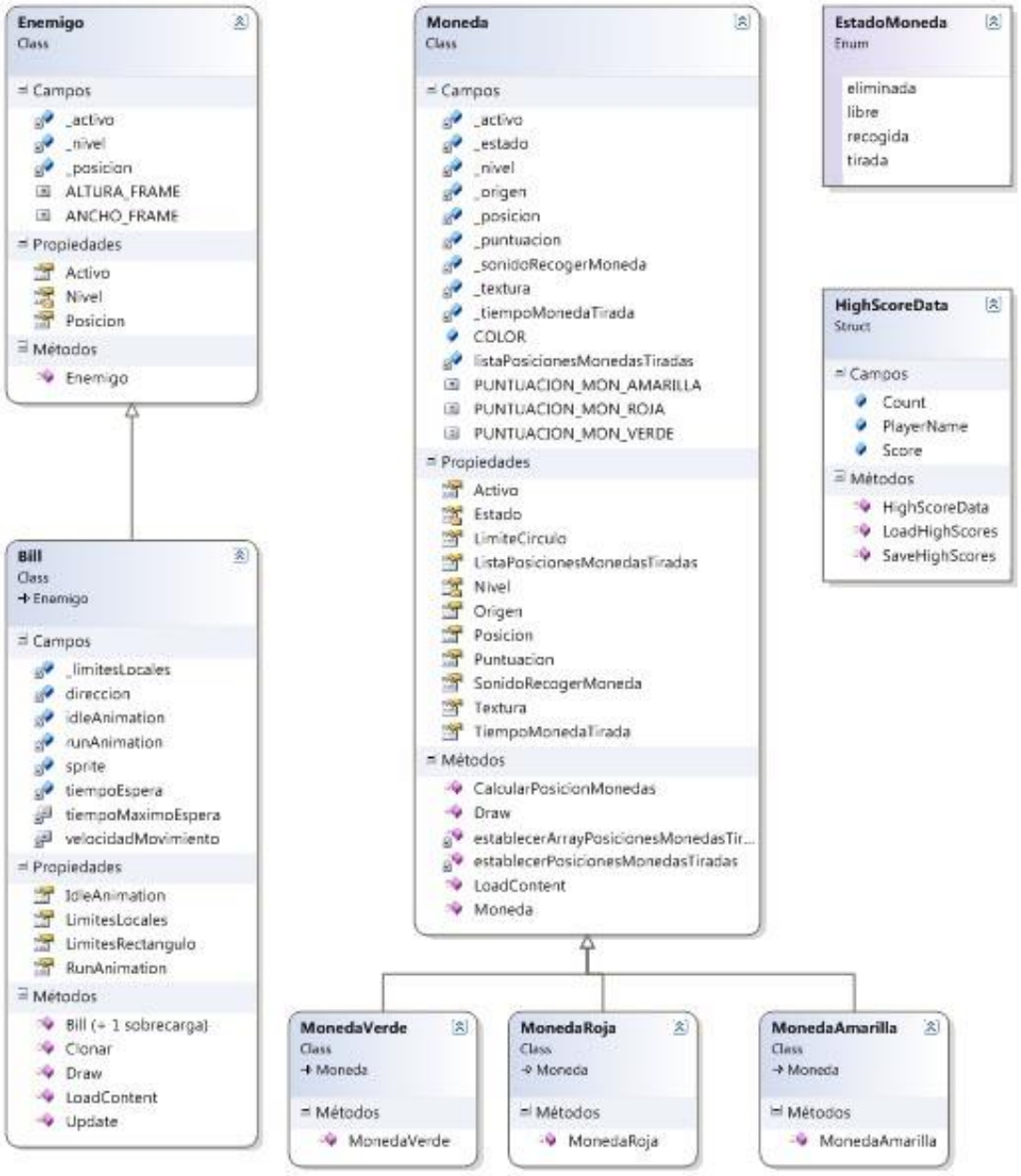


Ilustración 64 - Clases 2

Ilustración 65 - Clases 3

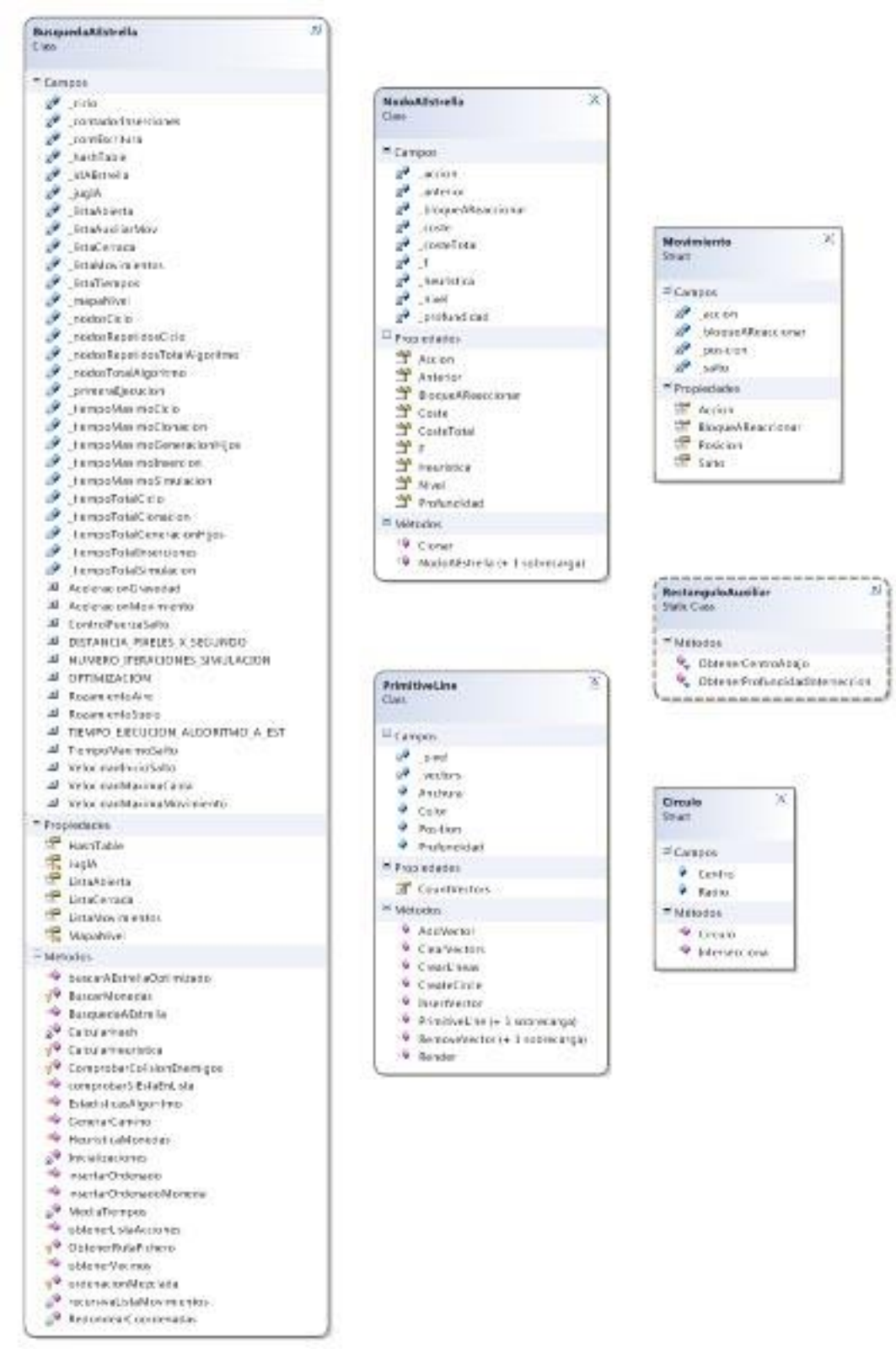


Ilustración 66 - Clases 4

4.3. Implementación

En esta sección se van a detallar los aspectos técnicos de la implementación del código del juego.

Para la escritura del código, se ha seguido la siguiente convención en cuantos a los nombres de las propiedades de las clases:

- *Propiedades privadas o protegidas* – Su nombre comienza por un guión bajo seguido de la primera letra en minúsculas. Todas las siguientes letras serán en minúscula también. Ejemplo: _jugador.

- *Propiedades públicas* – Su nombre comienza por la primera letra en mayúscula, y las demás en minúscula. Ejemplo: Jugador.

  Nota: Para ambas, en caso de estar formadas por dos palabras ó más palabras, se omitirá el espacio en blanco y la primera letra de la nueva palabra será en mayúsculas, quedando todas las demás letras en minúscula. Ejemplos: _jugadorSaltando ; EstaEnElSuelo

- *Constantes* – Todas las letras en mayúsculas. En caso de estar formado por varias palabras, se sustituirá el espacio en blanco por un guión bajo. Ejemplo: PUNTUACION_SEGUNDO

En cuanto a los métodos, todos sus nombres empezarán por mayúscula y todas las demás letras estarán en minúscula. En el caso de que el método estuviese compuesto por varias palabras, se omitirá el espacio en blanco y la primera letra de las siguientes palabras será en mayúsculas. Ejemplo: CargarMundoBloques.

4.3.1. Carga niveles

La carga del nivel es lo primero que se produce al iniciar una nueva partida. Para ello, se llama al constructor de la clase Nivel. Dentro de este constructor, se utiliza el método CargarMundoBloques(ficheroNivel) para cargar el contenido del fichero de nivel.

A dicho método se le pasa como parámetro el nombre del fichero a explorar. Este fichero se lee con un StreamReader, y va guardando cada línea en una lista de string.

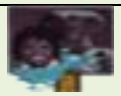
```
List<string> listaLineas = new List<string>();
            using (StreamReader reader = new StreamReader(ficheroNivel))
{
        linea = reader.ReadLine();
              …
 }
```

Lo primero que comprueba es la primera línea para obtener el tiempo de nivel, el cual ha de estar en formato M:SS, tal y como se describió en la sección de diseño. Una vez ha obtenido este dato, lo convierte a formato de tiempo y lo guarda en la propiedad _tiempoRestante del objeto.

```
_tiempoRestante = ConvertirATiempo(linea);
```

Posteriormente, se comienza el procesado de cada línea con formato de nivel. Un nivel está modelizado para ser representado por un array de N filas y M columnas, es decir, de 2 dimensiones. En cada uno de estos elementos, se carga su bloque correspondiente. Se recorre la lista de strings que contiene las líneas del fichero, y se va procesando carácter a carácter. Para este proceso unitario, se utiliza la clase *CargaTipoBloques(tipoBloque, x, y).* A este método se le pasa como parámetro el carácter leído, y la posición (x,y). Con el tipo de carácter, el método hace un filtro por medio de un switch para cargar el elemento que corresponda.

A continuación se muestra la lista completa de los elementos a cargar, así como su imagen correspondiente:

Carácter	Significado	Imagen
.	Bloque vacío (pasable)	
X	Bloque de salida (fin partida)	
A	Moneda Amarilla	
V	Moneda Verde	
R	Moneda Roja	
B	Enemigo	
1	Inicio jugador humano	
2	Inicio jugador Inteligencia Artificial	
@	Bloque mundo - Suelo	
#	Bloque mundo – Suelo verde	
$	Bloque mundo – Suelo césped 1	
%	Bloque mundo – Suelo césped 2	
&	Bloque mundo – Roca 1	
(	Bloque mundo – Roca 2	
)	Bloque mundo – Roca 3	
/	Bloque mundo – Pared	

Las imágenes del jugador humano, jugados del ordenador y los enemigos se muestran en la parte correspondiente a la animación de sprites.

Si se detectase un carácter distinto a los arriba especificados, la aplicación avisa del error.

Cada tipo de bloque tiene asociado otra función de carga característica por su tipología. Por ejemplo, los bloques de mundo simplemente cargan la textura y rellenan su posición en el array del mundo de bloques como noPasable. Los bloques de Inicio de Jugador y de inicio de jugadorIA, crean una nueva instancia del objeto jugador y jugadorIA respectivamente. Lo mismo ocurre con los enemigos. Para las monedas, cada vez que se detecta un bloque de moneda, se crea una instancia de dicha moneda y se incremente la lista de monedas de su tipo.

4.3.2. Animación sprites

La animación de sprites se realiza gracias a las clases Animacion y ReproduccionAnimacion. La primera es la encargada de almacenar las texturas de la animación. Dicha animación corresponde con un sprite, el cual tiene varias imágenes que se van sucediendo para dar el efecto del movimiento.

Para el jugador humano y el del ordenador existen los siguientes sprites:

- Movimiento y movimiento herido.
- Parado y parado herido.
- Salto y salto herido.
- Celebración.

La diferencia entre el sprite "normal" y el herido es que éste último se muestra con colores llamativos distintos, para indicar que el personaje se encuentra en estado herido (ha perdido monedas y permanece durante ese tiempo invulnerable).

Ejemplos de sprite de jugador:

Ilustración 67 - Sprite movimiento jugador

Ilustración 68 - Sprite salto jugador IA

Para el enemigo, el cual ha sido inspirado en la primera temporada de la serie de Enjuto Mojamuto, existen los siguientes sprites.

- Idle.
- Run.

El sprite Idle se utiliza cuando el personaje está parado en el tiempo de espera que realiza para cambiar el sentido de su marcha; y el sprite run se utiliza en su movimiento.

Ejemplo de sprite de enemigo:

Ilustración 69 - Sprite movimiento enemigo

Cuando se crea una animación, se carga el sprite que se desee, el tiempo que se reproduce cada frame, si es repite o no, y el número de frames que contiene el sprite:

```
                _jugadorSaltando = new
Animacion(Nivel.Content.Load<Texture2D>("Sprites/Jugador/Humano/salto"), 0.1f,
true, 4);
```

La clase ReproduccionAnimacion es la que se encarga de dibujar el sprite correspondiente. En función de la clase Animacion, conoce el tamaño del sprite y el número de frames que contiene. Dividiendo, y teniendo también en cuenta en todo momento el tiempo, puede conseguir la posición del sprite donde se encuentra el frame siguiente a ejecutar. Se hace el módulo entre el número de frames para que, al llegar al último frame, en la siguiente iteración se seleccione el siguiente. A continuación se muestra el código correspondiente al método Draw:

```
    // Proceso del tiempo transcurrido
    _tiempo += (float)gameTime.ElapsedGameTime.TotalSeconds;
    while (_tiempo > Animacion.TiempoFrame)
    {
       _tiempo -= Animacion.TiempoFrame;
       if (Animacion.EstaRepitiendose)
       {
           FrameIndice = (FrameIndice + 1) % Animacion.NumeroFrames;
       }
       else
       {
         FrameIndice = Math.Min(FrameIndice + 1, Animacion.NumeroFrames - 1);
       }
    }

    int anchoDeFrame = Animacion.Textura.Width / Animacion.NumeroFrames;
    // Calcula el rectángulo fuente del actual frame.
    Rectangle source = new Rectangle(FrameIndice * anchoDeFrame, 0,
anchoDeFrame, Animacion.Textura.Height);
     // Dibuja el frame actual.
     spriteBatch.Draw(Animacion.Textura, posicion, source, Color.White, 0.0f,
Origen, 1.0f, spriteEffects, 0.0f);
```

4.3.3 Cámara

La implementación de la cámara se llevó a cabo desde 0 ya que el juego Platform del Xna Creators no proporcionaba esta funcionalidad. El objetivo de la cámara es ir siguiendo al personaje del jugador humano.

Para realizar su diseño e implementación, se basó en la idea de que el jugador siempre tiene que estar centrado en la mitad de la pantalla. Sin

embargo, cuando el jugador está al principio o al final del nivel, no puede ser así porque ya no existe más nivel que mostrar. Este hecho hace, por tanto, que existan 3 posiciones o estados de cámara:

- 1er cuadrante: Inicio de nivel
- 2º cuadrante: Mediados de nivel
- 3er cuadrante: Final de nivel

La resolución del juego es de 800 píxeles, así que el ancho final del nivel se registra en la variable AnchoNivelEnPixeles multiplicando el número de bloques a lo ancho por la longitud del bloque.

```
_anchoNivelEnPixeles = Ancho * Bloque.Ancho;
```

Para calcular el cuadrante en el que se encuentra el jugador, y por tanto, la cámara que se ha de usar, se usa el método ObtenerCuadrante() con el siguiente algoritmo:

```
SI (POSICION_JUGADOR < 400)
        CUADRANTE = 1
SINO
        SI POSICION_JUGADOR < (ANCHONIVEL – 400)
                CUADRANTE = 2
SINO
        CUADRANTE = 3
```

Lo que se ha especificado en el algoritmo como POSICION_JUGADOR, en el código se denomina CamX, para que quedase más claro que se estaba trabajando con la finalidad de las cámaras.

La clasificación de la cámara está basada en la posición del jugador. Cuando el jugador se encuentra en el primer o tercer cuadrante, la cámara está fija. Sin embargo, cuando el jugador se encuentra en el segundo cuadrante, la cámara se mueve en función de dicha posición.

La finalidad de calcular en qué cuadrante se encuentra es para el método de dibujo. Es muy importante este dato para saber a qué altura dibujar todos los elementos del juego, tanto el nivel, como el jugador, los enemigos, etc. Si el cuadrante es el primero, el origen sería el (0, 0). En este punto es preciso recordar que en Xna las coordenadas tienen su origen en la esquina superior izquierda. Por tanto, se calcularían todas las posiciones a partir de dicho origen.

Sin embargo, si se encuentra en el cuadrante segundo o tercero, el origen es distinto. Por ejemplo, al dibujar el jugador, si está en el segundo cuadrante se dibuja con la siguiente instrucción (Clase: Jugador ; Método: Draw) :

```
_sprite.Draw(gameTime, spriteBatch, new Vector2(400, Posicion.Y), _voltear);
```

De esta manera, el jugador siempre se dibuja en el centro de la pantalla, es decir, en el punto 400 del eje X (ya que la resolución de pantalla es de 800). La posición en el eje Y es la que le correspondiese por el estado del juego.

Si, por el contrario, el jugador se encontrase en el último cuadrante, habría que calcular su posición y pintarlo, con el siguiente código:

```
float nuevaPosicionX = 800 - (Nivel.AnchoNivelEnPixeles - Posicion.X);
sprite.Draw(gameTime, spriteBatch, new Vector2(nuevaPosicionX,
Posicion.Y), _voltear);
```

La variable nuevaPosicionX calcula la posición en la que estaría en función de la resolución del ancho de la pantalla, el ancho del nivel y la posición en el mismo del jugador.

Con el jugador del ordenador (IA) no hace falta hacer este seguimiento de la cámara. Simplemente, hay que calcular en qué posición está el jugador del humano para ver si hay que mostrarle o no, porque se encuentre en sus cercanías. Encontrarse "cerca" significa estar, como máximo, a 400 lados a ambos lados del personaje humano, ya que la resolución es de 800 píxeles. Si, efectivamente, es así, hay que calcular la posición en la que se dibujará al personaje de la IA en función del cuadrante

en el que se esté, para posicionar la textura. Todo ello se hace siguiendo el siguiente código:

```
// Depende en todo momento de la posición del jugador humano (Jugador.Pos)
if (pPosicionJugador.X < 400) //El humano está en el 1er cuadrante
{
    //Comprobación de que la IA esté entre el 0 y el 800 para pintarle
    if (Posicion.X < 800)
    {
          sprite.Draw(gameTime, spriteBatch, Posicion, _voltear);
    }
 }
else if (pPosicionJugador.X > Nivel.AnchoNivelEnPixeles - 400)  //Jugador humano en 3er cuadrante
      {
         //Comprobación de que la IA esté al final
          if ((Posicion.X >= Nivel.AnchoNivelEnPixeles - 800) && (Posicion.X <= Nivel.AnchoNivelEnPixeles))
         {
              float nuevaPosicionX = 800 - (Nivel.AnchoNivelEnPixeles - Posicion.X);
              sprite.Draw(gameTime, spriteBatch, new Vector2(nuevaPosicionX, Posicion.Y), _voltear);
          }
      }
       else //el humano está en el segundo cuadrant
      {
       //comprobación de que la IA esté en el segundo cuadrante y a una distancia de al menos 400 en ambos sentidos del humano
           if ((Posicion.X >= pPosicionJugador.X - 400) && (Posicion.X <= pPosicionJugador.X + 400))
           {
              float nuevaPosicion = Posicion.X - pPosicionJugador.X;
              sprite.Draw(gameTime, spriteBatch, new Vector2(400 + nuevaPosicion, Posicion.Y), _voltear);
           }
      }
```

Con las monedas, sucede algo parecido. Se calcula el cuadrante, y en función de ello se dibuja cada moneda. Si está en el primer cuadrante, en su posición con normalidad. Si está en el segundo, se resta a la posición del jugador (CamX) la mitad de la resolución de pantalla (400) y se le cambia el signo:

```
spriteBatch.Draw(_textura, new Vector2(-((Nivel.CamX - 400) - Posicion.X), Posicion.Y), null, COLOR, 0.0f, _origen, 1.0f, SpriteEffects.None, 0.0f);
```

Con el tercer cuandrante, hay que restar en el eje X a la posición del jugador el ancho del nivel – la anchura de su resolución:

```
                spriteBatch.Draw(_textura, new Vector2((Posicion.X -
(Nivel.AnchoNivelEnPixeles - 800)), Posicion.Y), null, COLOR, 0.0f, _origen, 1.0f,
SpriteEffects.None, 0.0f);
```

En cuanto al dibujo de los niveles, es importante calcular la posición del respecto al nivel para pintar los bloques correspondientes. Si se está en el primer o tercer cuadrante, se pintan los bloques del principio de la matriz o del final, respectivamente, con recorrido simple de dos bucles for. Sin embargo, si se está en el cuadrante intermedio, es más complicado porque hay que llevar la transición correcta entre el dibujo de un bloque y el siguiente. Es decir, que al pintar los bloques de una columna, no haya "saltos" al pasar del bloque i al bloque i+1, sino que vaya en progresión el dibujo del bloque. Además, hay que calcular los bloques que se han de dibujar, tanto a un lado del jugador como al otro. Todo ello se consigue con el siguiente código:

```
        //Intermedio
                int bloqueEnElQueEstaElJugador = CamX / Bloque.Ancho;
                int ejeX = 0;
                int ejeY = 0;

                int minimoEjeX = Math.Min(bloqueEnElQueEstaElJugador + 11,
_bloques.GetUpperBound(0));

                for (int y = 0; y < Alto; ++y)
                {
                   int incremento = ((CamX - 400) % Bloque.Ancho);
                   ejeX = 0;
                   for (int x = bloqueEnElQueEstaElJugador - 10; x <= minimoEjeX;
++x)
                   {
                      // If there is a visible tile in that position
                      Texture2D texture = _bloques[x, y].Textura;
                      if (texture != null)
                      {
                         // Draw it in screen space.
                         Vector2 position = new Vector2(ejeX, ejeY) * Bloque.tamanio;
                         position.X -= incremento;
                         spriteBatch.Draw(texture, position, Color.White);
                      }
                      ejeX++;
                   }
                   ejeY++;
                }
```

4.3.4 Parallax scrolling

El parallax scrolling son las imágenes de fondo del juego que se van moviendo para dar sensación de mayor desplazamiento. Se compone de varias capas que se suporponen unas a otras. Cada una de ellas puede llevar una velocidad distinta. Para mantener en el juego varias capas, se tiene una lista de capas.

Cada una de estas capas es del tipo CapaParallax. Se crean en la función *CargarCapasParallas(Viewport pViewport)*y se añaden a la lista de capas (_listaCapasParallax). En su constructor, se le indica la textura, el ancho de la ventana (Viewport.Width); y se inicializa la posición y el desplazamiento. Por último, se le asigna también la velocidad, la cual se especificó al crear la capa en el último parámetro

Se han recreado dos escenificaciones para estos fondos: uno simula que la partida se desarrolla por el día; y otro en el que se desarrolla por la noche. En la de día, se carga una primera capa con el cielo despejado, y luego dos capas con nubes blancas y nubes grises. La primera capa de cielo es fija, mientras que las de las nubes se mueven a diferentes velocidades. Por su parte, la simulación de noche consta de un cielo oscuro, de otro con la luna y otra más con nubes. En este caso, las capas con movimiento son la luna y las nubes. A continuación se muestran dos imágenes de ambos escenarios:

Ilustración 70 – Parallax-scrolling día

Ilustración 71 - Parallax-scrolling noche

La selección de uno u otro escenario se lleva a cabo con total aleatoriedad. Se ha hecho de esta manera aleatoria para dotar al juego de mayor diversión, al no ser siempre igual. Para ello, en la función CargarCapaParallax, se obtienen los milisegundos del tiempo que lleva la aplicación corriendo. Si su resto tras dividirlo entre 2 es 0 (es decir, es un número par), se carga el escenario de noche; en caso contrario, se carga el escenario de día.

En el método Update de CapaParallax, lo único que se hace es llamar al método MoverDerecha. Éste se encarga de mover los frames de acuerdo a la velocidad que se indicó en la creación del objeto.

En cuanto al dibujo, se dibuja por pantalla como una textura normal; pero, en la posición, se suma el desplazamiento a la posición. Éste desplazamiento viene determinado en función de la anchura de la pantalla. En este método se lleva a cabo también el reinicio de la textura si se ha visualizado por completo. Indicar también que es importante el orden en que llamar al método dibujo de cada capa. Para que se pinten correctamente las superposiciones una sobre otra, hay que llamar primero a la de más al fondo, y así sucesivamente.

4.3.4. Colisiones

El sistema de colisiones tiene lugar en el método *AplicarFisica* de las clases Jugador y JugadorIA. Es el encargado de comprobar el apartado físico, como que por ejemplo exista gravedad o que no se puedan traspasar paredes. El funcionamiento del método está basado en la posición del jugador. Para calcular su posición, se utiliza la propiedad velocidad, la cual es calculada por medio de las constantes de la clase y del movimiento del jugador (si ha pulsado las teclas de dirección o de salto, la cual es recogida con el método *ObtenerEntradaTeclado*). Se calculan por separado el eje X y el eje Y para hacer posteriormente una suma vectorial. También se tiene en cuenta los factores de rozamiento. Si el jugador salta, para calcula la física se llama al método Saltar().

```
Posicion += Velocidad * tiempoTranscurrido;
```

Una vez que la posición del personaje se ha actualizado, se procesan las colisiones mediante la llamada al método ComprobarColisiones. El funcionamiento de este método es el siguiente:

Se obtiene el bloque en el que se encuentra el jugador, a partir de su posición. A partir de este bloque, se obtienen (gracias a la matriz que contiene el mundo de los bloques, todos los bloques adyacentes, tanto en el eje horizontal como el vertical. Todos esos bloques son los que van a ser procesados; pues, evidentemente, son con los únicos con los que el jugador podría colisionar. Para dicha verificación, se comprueba el tipo de bloque que es. Si es un bloque pasable, se continua con el siguiente a procesar. Sin embargo, si es un bloque noPasable, hay que comprobar si realmente el frame del jugador colisiona con dicho bloque. Para ello, se crea un rectángulo auxiliar, con los límites de los rectángulos del frame del jugador y del bloque. Con dicho rectángulo, se llama a la función *ObtenerProfundidadInterseccion*, la cual devuelve 0 si no hay intersección, y en caso contrario, devuelve la profundidad de la intersección. Gracias a este valor de profundidad, se puede resolver si el personaje se tiene que parar porque colisione o no, obteniendo el rectángulo que contiene al frame del personaje resultado:

```
        Vector2 profundidad =
RectanguloAuxiliar.ObtenerProfundidadInterseccion(limitesRect, limitesBloque);
        if (profundidad != Vector2.Zero)
        {
        float absDepthX = Math.Abs(profundidad.X);
        float absDepthY = Math.Abs(profundidad.Y);

        // Resuelve la colisión a lo largo del eje X (plano)
        if (absDepthY < absDepthX || colision == colisionBloque.plataforma)
        {
         // Si cruzamos la parte de arriba de un bloque, estamos en el suelo
            if (_fondoAnterior <= limitesBloque.Top)
            {
                 _estaEnElSuelo = true;
            }
            //Ignoramos plataformas, a no ser que estemos en el suelo
            if (colision == colisionBloque.noPasable || EstaEnElSuelo)
            {
               // Resolvemos las colisiones a lo largo del eje Y
                Posicion = new Vector2(Posicion.X, Posicion.Y + profundidad.Y);
               // Perform further collisions with the new bounds.
               limitesRect = LimitesRectangulo;
            }
        }
           else if (colision == colisionBloque.noPasable) // Ignoramos plataformas
           {
               // Resolvemos las colisiones a lo largo del eje X
               Posicion = new Vector2(Posicion.X + profundidad.X, Posicion.Y);
               // Perform further collisions with the new bounds.
               limitesRect = LimitesRectangulo;
            }
    }
```

4.3.5 Tirar monedas

Cuando un enemigo toca al personaje, tanto el del humano como el del ordenador, éste pierde monedas. Si el personaje del jugador humano no tuviese monedas, la partida finalizaría. La pérdida de monedas se lleva a cabo mediante un lanzamiento de las monedas en las posiciones cercanas. El encargado de este proceso es el método TirarMonedas(), de la clase Moneda.cs.

Cuando se detecta una colisión del personaje con el enemigo, se llama al método TirarMonedas(). Éste, lo primero que hace, es calcular el número de monedas a tirar, con el método *CalcularMonedasATirar()* de la clase Jugador. Como máximo se desprenden de 12 monedas: 6 amarillas, 4

verdes y 2 rojas, para llevar una proporcionalidad justa con su valor. En el caso que no tuviese suficientes monedas de un tipo, la diferencia entre lo que debería haber tirado y lo que realmente tira se suma al siguiente tipo de moneda; y así, sucesivamente. Se usa una variable llamada contadorMonedasQueFaltan para acumular el número de monedas que están "en deuda" por ser tiradas.

Por ejemplo, si el jugador tuviese 4 monedas amarillas, 3 verdes y 8 rojas, arrojaría en total las 4 amarillas, las 3 verdes y 7 rojas. Tira 6 rojas, 4 más de las que debería haber lanzado "por defecto", ya que no tenía suficientes monedas amarillas, le faltaban 2. Éstas, se intentaron sumar a las verdes por defecto para lanzar, pero tampoco tenía suficientes de este tipo. Por tanto, se sumarían las 2 amarillas del principio más las 3 verdes que le faltan. Por tanto, al llegar a calcular el número de rojas de las que ha de desprenderse, el resultado da 7, que es la suma de las 2 por defecto rojas, más el acumulado de amarillas (2) y verdes (3).

En el caso de que no se dispusiese de monedas suficientes a tirar según el anterior algoritmo, se tirarían todas las monedas recogidas y el jugador se quedaría sin ninguna.

El método CalcularMonedasATirar() devuelve un array de enteros con la cantidad a monedas de cada tipo a tirar. En el primer registro se devuelven las amarillas, en el segundo las verdes y en el tercero las rojas. Con estos valores ya calculados, el método TirarMonedas continúa calculando las posiciones a donde se lanzarán. Para ello, utiliza el método CalcularPosicionesMonedas(). Este método se ocupa de distribuir las monedas uniformemente en forma de semicircunferencia alrededor del jugador, por la parte superior. Para ello, se ayuda de una lista de posiciones relativas que marcan los vectores posición a de dicha forma, tal y como muestra la siguiente figura:

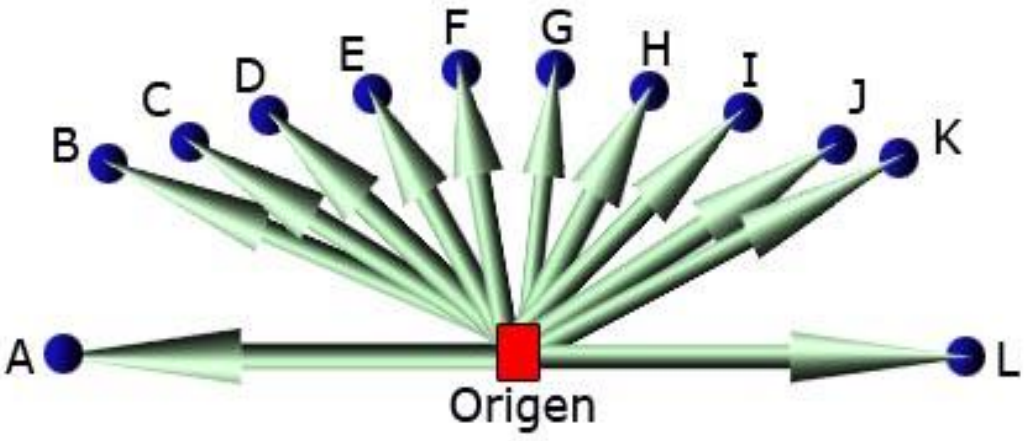


Ilustración 72 - Posiciones lanzamiento monedas

Los vectores correspondientes a casa punto son los siguientes:

Punto	Posición
A	(-187.46;37.5)
B	(-162.48; 112.5)
C	(-132.44;117.22)
D	(-94.6; 126.65)
E	(-56.76;136.08)
F	(-19.92;145.51)
G	(19.92; 145.51)
H	(56.75; 136.08)
I	(94.6; 126.65)
J	(132.44; 117.22)
K	(162.48; 112.5)
L	(187.46; 37.5)

Con esta lista de vectores se consigue calcular a qué posición se lanzaría la moneda, con la suma vectorial a la posición del jugador.

La distribución de las monedas se haría uniformemente según el número de monedas a lanzar.

A partir de ahí, se van rellenando las posiciones con las monedas amarillas, verdes y rojas, en respectivo orden. Por cada moneda tirada, la lista que almacena las monedas recogidas se decrementa en una unidad con dicha moneda, y la lista que almacena las monedas tiradas se incrementa. Se sigue este proceso para todos los tipos de monedas. Las monedas tiradas permanecen en pantalla 5 segundos. A partir de ese momento, desaparecen. Para controlar y gestionar el tiempo que llevan, se utiliza el método *UpdateMonedasTiradas()*.

4.3.5 Pintar líneas simulación IA

La simulación del movimiento del personaje IA se pinta con una línea roja. De esta manera, se puede saber cuál es la predicción que ha hecho el algoritmo sobre cuál debería ser el movimiento en los instantes posteriores del personaje controlado por el ordenador. Esta opción se añadió también para depurar el algoritmo.

Para esta labor se utiliza la clase PrimitiveLine. Esta clase está compuesta por una lista de vectores. A partir de los puntos que representan los vectores, se generan líneas. Estas líneas están caracterizadas por las propiedades de la clase Color, Profundidad y Anchura. En el constructor de la clase se inicializan todas estas propiedades, así como se asigna la propiedad pixel, que es la textura que se pintará en las líneas.

Con el método CrearLineas(List<Vector2> pListaVectores) se añaden todos los puntos que formarán las líneas. Lo que hace es rellenar la propiedad de la clase vectors con la lista que se pasa por parámetro.

Finalmente, con el método Render se pinta la línea. Este método se encarga de generar las líneas, tomando un par de puntos consiguientes de la lista vectors mediante un bucle for. Con estos dos puntos, crea un vector distancia con su resta y calcula el ángulo que formarían; para poder dibujarlo con la instrucción:

```
                spriteBatch.Draw(pixel, Position + vector1, null, Color, angle,
Vector2.Zero, new Vector2(distance, Anchura), SpriteEffects.None, Profundidad);
```

El primer parámetro representa a la textura que se dibuja. En el segundo parámetro la posición. El tercer parámetro es nulo porque sería el rectángulo donde se dibujaría, pero en este caso no aplica. El cuarto parámetro es el color, previamente fijado en el constructor. El quinto parámetro la rotación de la textura, a la cual se asigna el ángulo entre los dos puntos anteriormente calculado. El sexto parámetro es el origen, pero no este caso no hace falta ponerlo. El siguiente parámetro, séptimo, corresponde a la escala, la cual se calcula en función de la distancia y la anchura de la línea. Por último, no se dibuja con ningún efecto particular y se le asigna una profundidad a la capa de 0 para que esté al frente.

En la siguiente imagen se muestra un ejemplo del resultado final:

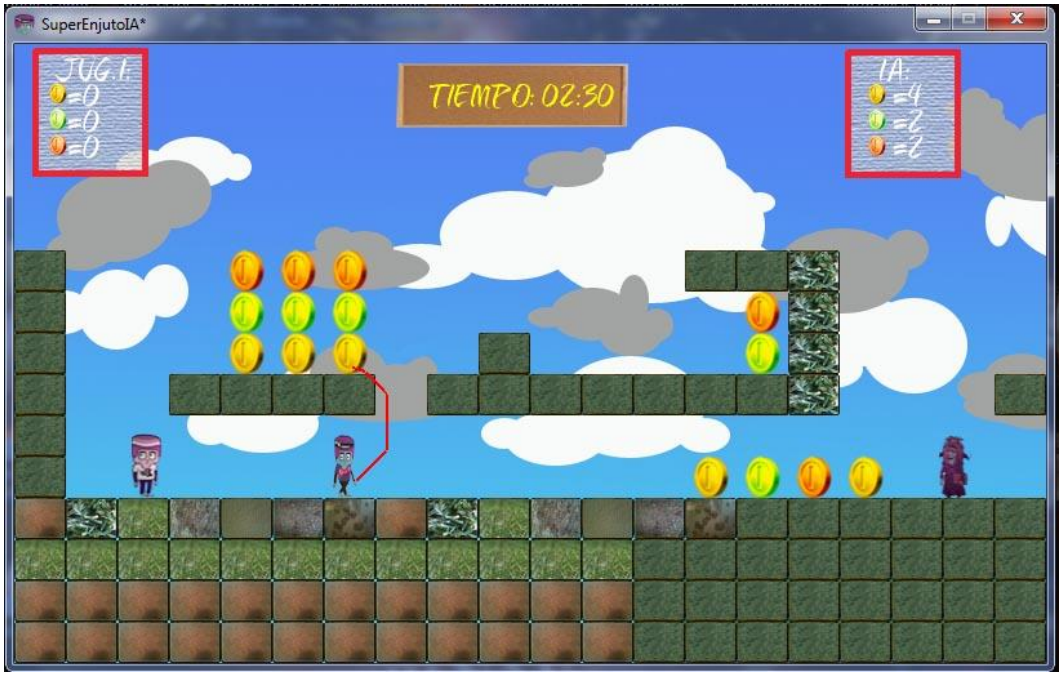


Ilustración 73 - Líneas simulación IA

4.3.6 Inteligencia Artificial

Antes de entrar a la explicación técnica de cómo se ha implementado el algoritmo en este proyecto, se explica en qué consiste y su funcionamiento general, para una mayor comprensión:

4.3.6.1 A*

La técnica de Inteligencia artificial utilizado para el personaje del ordenador es la búsqueda de caminos con el **algoritmo A***. Si existe un camino viable entre dos puntos, este algoritmo lo encuentra siempre. Por tanto, es un algoritmo *óptimo.* No solo encuentra el camino, sino que si la heurística utilizada es admisible (mejor al coste real para llega al nodo meta), entonces el algoritmo devuelve dicho camino óptimo. Es decir, que si al algoritmo se le da el tiempo suficiente para que se ejecute completamente y tiene una heurística admisible, encuentra siempre el camino. Este hecho no es 100% garantizable en este proyecto porque no se cumplen siempre los requisitos temporales, ya que se interrumpe la ejecución del algoritmo por cuestiones de eficiencia.

El camino a recorrer se suele dividir y representar en estructuras de nodos, pudiendo tener forma de grafo o de árbol. Surge como extensión del algoritmo de Dijkstra.

A* se basa en una combinación de búsquedas del tipo "primero en anchura" y "primero en profundidad". La función que representa el algoritmo es:

$$f(n) = g(n) + h(n)$$

La parte de la ecuación referente a la búsqueda *primero en anchura* es g(n); y es el coste real del camino recorrido hasta dicho punto. La búsqueda de *primero en profundidad* viene dado por el sumando h(n); y representa el valor heurístico (estimado) desde el punto actual hasta el punto destino.

El algoritmo también consta de dos estructuras auxiliares: *lista abierta* y *lista cerrada*. En la primera se van almacenando los nodos por explorar, es decir, los que todavía no han sido procesados. Sobre esta lista se ha de añadir también que está ordenada por el valor del nodo, quedando los que

tienen un valor menor en primer lugar. *Gracias a esta ordenación el camino encontrado es el óptimo.* Por otro lado, la lista cerrada es la encargada de almacenar los nodos ya tratados.

El mecanismo de funcionamiento del algoritmo se define de la siguiente forma: Dado un nodo (al principio será el nodo inicial), se crean todos los nodos "vecinos". Un nodo vecino es aquel al que se puede acceder desde el nodo actual. Para completar el proceso de creación de éstos, se han de evaluar dichos nodos. La evaluación consiste en aplicar la fórmula anteriormente descrita (f(n) = g(n) + h(n)), siendo g(n) el coste acumulado hasta tal nodo y la estimación hasta el final. Cada nodo tendrá sus propios valores, según sus características. Por ejemplo, el paso de un nodo a otro pueden tener costes distintos, y la heurística (valor estimado de coste hasta llegar al final) puede ser distinta entre ambos. Con estos valores calculados, se suman y se obtiene el valor total del nodo, es decir, f(n). Una vez que el nodo ha sido creado, se inserta en la lista abierta, ordenado por su valor total. Para que la inserción se produzca, el nodo no debe estar repetido, es decir, no tiene que estar en la lista abierta ni en la lista cerrada. Si ése fuera el caso, se descartaría automáticamente.

Al acabar la fase de creación de nodos y su posterior inserción en la lista abierta, se produce la fase de inserción en la lista cerrada. A esta lista va a parar el primer nodo de de la lista abierta, es decir, se extrae de abierta para introducir el nodo en la lista cerrada. De este modo se consigue haber seleccionado el nodo con mejor puntuación, ya que recordemos que la lista abierta estaba ordenada de menor valor a mayor. Se considera que cuanto mayor es el valor, peor resultado es, ya que el coste siguiendo ese camino sería mayor.

Todo el tratamiento anterior se plasma en el siguiente pseudocódigo:

```
ABIERTOS := [INICIAL] //inicialización
CERRADOS := []
f'(INICIAL) := h'(INICIAL)
repetir
        si ABIERTOS = [] entonces FALLO
        si no // quedan nodos
```

```
        extraer MEJORNODO de ABIERTOS con f' mínima
        // cola de prioridad
        mover MEJORNODO de ABIERTOS a CERRADOS
        si MEJORNODO contiene estado_objetivo entonces
        SOLUCION_ENCONTRADA := TRUE
        si no
        generar SUCESORES de MEJORNODO
        para cada SUCESOR hacer TRATAR_SUCESOR
hasta SOLUCION_ENCONTRADA o FALLO
```

A partir de este punto, se volvería a repetir el ciclo a partir del siguiente nodo con valor más bajo en la lista de abiertos. Es decir, se seleccionaría el primer nodo de la lista abierta y se calcularían todos sus vecinos. Este proceso se repetiría constantemente hasta llegar alcanzar el nodo final. Por tanto, antes de generar todos los nodos hijos, se ha de comprobar si el nodo actual corresponde al nodo final, para concluir el algoritmo.

Cuando se ha alcanzado el objetivo, el último paso que hay que hacer es obtener la lista completa de movimientos. Este paso se puede realizar de un modo sencillo escalando el árbol desde dicho nodo al nodo inicial. Dicho proceso es posible porque, cada vez que se genera un nodo vecino, o mejor dicho hijo, se va guardando con un puntero quién es su padre. De esta forma, se va obteniendo el camino trazado desde el nodo inicial al nodo final.

Es importante destacar que la fortaleza, así como el factor de éxito, de este algoritmo reside en su heurística. Calcular el coste que supondría la transición desde un estado a otro es trivial; sin embargo, el diseño de su heurística no es una tarea tan sencilla, ya que es un proceso de estimación, y por tanto, no tiene una garantía asegurada. Depende, por tanto, subjetivamente del analista encargado de realizar la estimación y su grado de acierto/fracaso.

Algunas heurísticas bastante utilizadas comúnmente son:

- Distancia Euclídea – Es la distancia que hay desde el punto actual hasta el punto final en línea recta. Por tanto, se da por hecho,

que el jugador puede avanzar en todas las direcciones del plano, incluyendo las oblicuas.

- Distancia Manhattan – Es la distancia que habría desde el punto actual hasta el punto final si los movimientos sólo pudiesen ser horizontales o verticales.

A continuación, se expone un problema de búsqueda de caminos y cómo quedaría el grafo de resolución utilizando el A*:

Teniendo el mapa que se muestra en la siguiente figura, el objetivo es ir desde el nodo inicial (11) hasta el nodo final (14). Se puede mover sólo horizontalmente y verticalmente, es decir, que no puede realizar movimientos oblicuos.

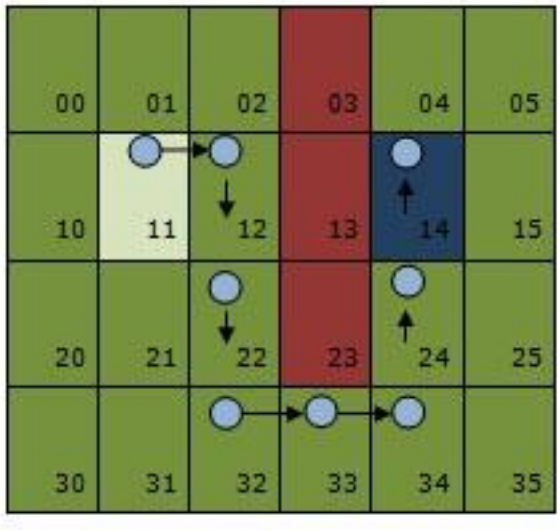


Ilustración 74 - Ejemplo tablero A*

Si se tiene en cuenta que el movimiento desde una casilla a otra colindante es siempre el mismo, y que la heurística que se usa es la de la distancia Manhattan, se tiene el siguiente grafo resultado:

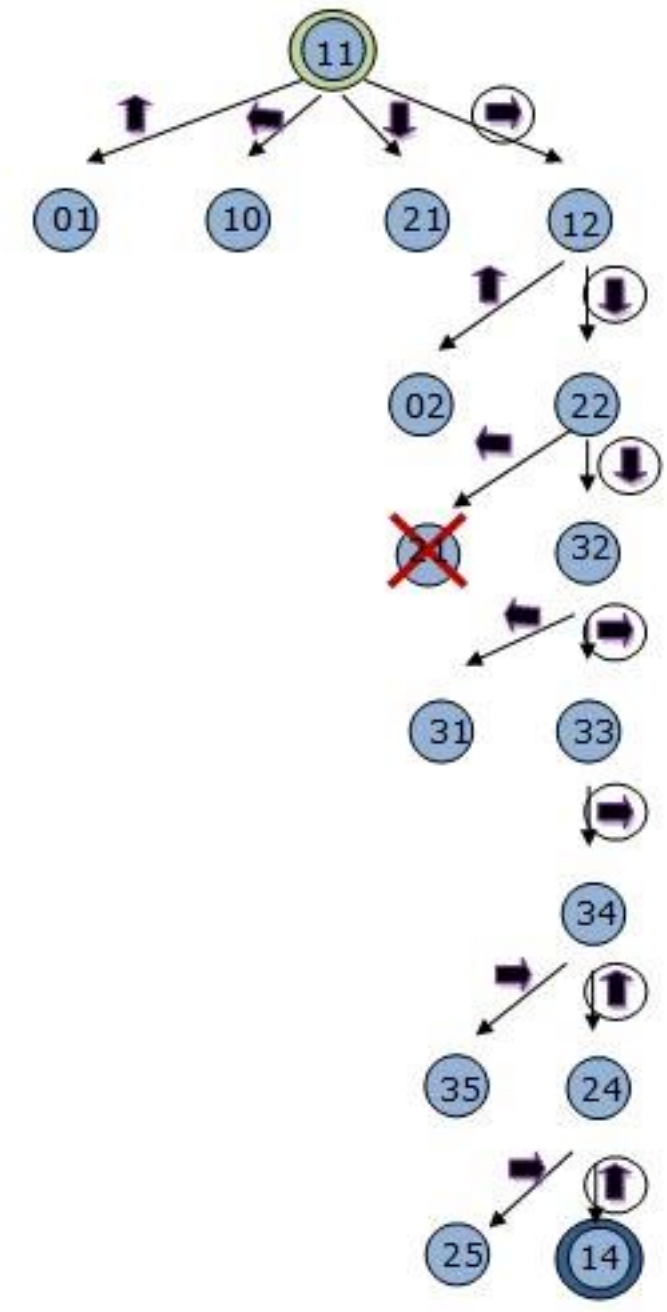


Ilustración 75 - Ejemplo árbol A*

No es el propósito de este sub-apartado explicar el algoritmo aplicado al ejemplo en exactitud y detalle, pero sí se quieren destacar varios aspectos y consideraciones que son importantes para el entendimiento de la técnica:

- Se crean nodos siempre que sea posible. Por ejemplo, en la posición 11 se crean los 4 nodos correspondientes a todas las direcciones ya que no hay obstáculos. Sin embargo, en la posición 12 sólo se crean 2 nodos ya que la pared impide ir hacia la derecha (el otro no que falta es el de la posición 11, el cual no se pone porque ya está en la lista cerrada habiendo pasado por la abierta).

- Cuando en el grafo se representa el nodo 21 tachado, significa que no se procesa porque ya había sido tratado antes. Esto, en traducción al código del algoritmo, significaría que el nodo 21 ya está introducido en la lista abierta, por lo tanto no se volvería a introducir. Nota: En el grafo no se han representado tampoco los nodos de los que se procede. Es decir, al generar sus vecinos, también se genera el nodo del que se viene, pero éste posteriormente no se introduce en la lista abierta porque ya está en la cerrada. No se ha dibujado en la figura para que quede más claro el diagrama.

- Si dos nodos tuviesen el mismo valor, a la hora de ordenar en la lista abierta daría igual cual se pusiese delante y después.

4.3.6.2 Aplicación a Super Enjuto IA *

Toda esta explicación corresponde al tratamiento general del algoritmo A*. Sin embargo, para aplicarlo a este proyecto en concreto, ha habido que hacer una serie de tareas en función de la naturaleza del problema que aquí se trata para una correcta implantación.

En el ejemplo anterior, el nivel es un mapa bidimensional estructurado en sectores o casillas. Este hecho se podría aplicar perfectamente a juegos de estrategia como de la imagen 78, porque el modo de juego este género de videojuegos se basa en mapas de este tipo. Se usaría, por ejemplo, para mandar a un escuadrón de guerreros a atacar al enemigo; y con este

algoritmo buscaría el camino que debería seguir evitando posibles obstáculos.

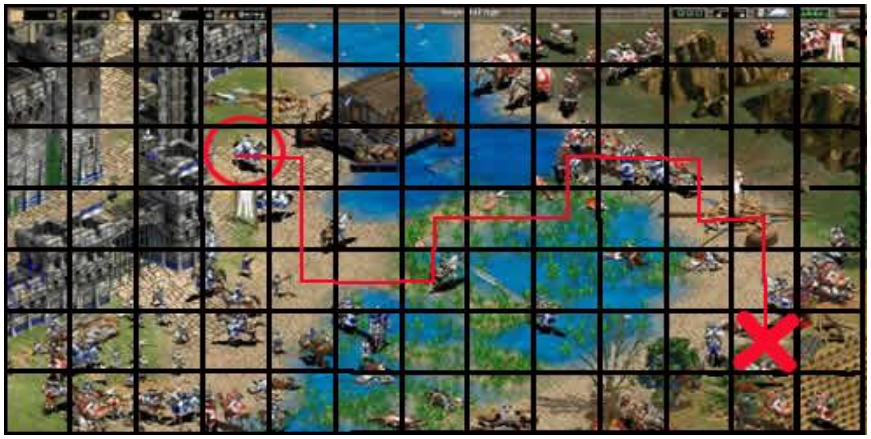


Ilustración 76 - Ejemplo A* en juego de estrategia

Sin embargo, el juego que aquí nos ocupa es de otro género completamente distinto, las plataformas, donde esta modelización no es posible tal cual. En primer lugar, porque la vista del nivel no es cenital (como en los juegos de estrategia) sino que es frontal al personaje y al nivel. Nos enfrentamos, por tanto, a un cambio de perspectiva el cual hay que modelizar correctamente. El detalle más importante ante este cambio es la inclusión de la gravedad, la cual hay que tener muy en cuenta, pues tiene un efecto directo en los desplazamientos, tanto en el eje de abscisas como en el eje de ordenadas.

En este punto es importante destacar que, aunque el juego está estructura en celdas, el movimiento del personaje no es discreto, es decir, no pasa de una celda a otra directamente; sino que el personaje va moviéndose progresivamente. Esto impide realizar un A* "modélico" con transiciones entre casillas.

El de colisiones, también se genera en base a la matriz bidimensional de bloques. Por ejemplo, imaginemos que el jugador está en la posición A, y que el centro del frame corresponde al bloque [12,10]. En el mapa de nivel, el cual guarda el array bidimensional con la morfología de cada bloque (si es pasable o no, es decir, si es un obstáculo), podremos comprobar las

colisiones que habría con sus bloques vecinos. Pero, para determinar si el jugador llega al bloque siguiente, hay que hacerlo en progresión con las ecuaciones del movimiento que lleve dicho personaje, lo cual se hablará más tarde en este apartado. Además, de esta forma se han de tener en cuenta en la simulación aspectos de la física: por ejemplo, que se esté apoyado en el suelo, que se salte y se choque con una plataforma o la inercia del movimiento.

El método *EntradaAccionBot(GameTime gameTime, Nivel pNivel)*, perteneciente a la clase JugadorIA, es el encargado de encontrar el camino a recorrer por el personaje del ordenador, lanzando el algoritmo A*. Este hecho se produce siempre y cuando el personaje IA no haya llegado al final del nivel, pues ya no haría falta. Por tanto, dentro de este método se llama a la función *GenerarCamino,* de la clase BusqAEstrella. Esta función, la cual contiene todo el algoritmo A* y el cual se explicará a continuación, devuelve una lista con los movimientos que se han obtenido. Esta lista está compuesta por estructuras del tipo Movimiento, la cual está formada por:

- Accion: Entero que representa la acción que realiza el jugador, siendo 1 movimiento hacia la derecha; 3 movimiento hacia la izquierda y 0 si no realiza desplazamiento.
- BloqueAReaccionar: Entero que representa el bloque del ejeX en el cual esa acción se realiza.
- Salto: Boolean que controla si el personaje acciona el salto o no.
- Posicion: Posición (Vector2) del jugador cuando realiza la acción.

El elemento más importante de esta lista es el primero (_listaMovimientos[0]) porque es la acción que el jugador de la IA va a realizar. Sin embargo, se guarda la lista completa de movimientos para poder dibujar con una línea roja la trayectoria que ha simulado el algoritmo, tal y como se explica en el sub-apartado *Pintar líneas de simulación*.

Para instar al personaje a que realice la opción calculada, se obtiene el primer elemento de la lista de movimientos y se van asignando todas sus propiedades al jugadorIA:

```
this.Accion = registroMovimiento.Accion;
switch (this.Accion)
      {
            case (int)Acciones.derecha:
                  Movimiento = 1.0f;
                  break;

            case (int)Acciones.izquierda:
                  Movimiento = -1.0f;
                  break;

            case (int)Acciones.parado:
                  Movimiento = 1.0f;
                  break;
      }

if (registroMovimiento.Salto == true)
      {
          EstaSaltando = true;
      }
      else
      {
          EstaSaltando = false;
      }
```

En el caso que el jugador hubiese llegado a la meta, comprobación la cual se realiza antes de lanzar el algoritmo A*, el personaje de la IA cambia su estado a final y se para.

Se decía que el elemento más importante era el primero porque es el que va a ejecutar el personaje. La idea es que el algoritmo se vaya lanzando cada 3 ciclos de actualización del juego (Update) ó ticks, con una técnica de balanceo del algoritmo que se explicará más adelante.

Si el algoritmo fuese lanzado sólo una vez al principio, no sería capaz de captar los cambios de estado del nivel. En ese hipotético caso, se simularía el mapa de nivel del principio y se obtendría el camino a recorrer en función de éste. No sería necesario lanzar nuevamente el algoritmo, porque ya tendría el mejor camino y simplemente tendría que ejecutar todas sus acciones.

Sin embargo, este no es el caso de este juego, porque el estado del nivel cambia en tres factores fundamentales:

- Enemigos: Los enemigos se mueven horizontalmente hasta que se topan con un obstáculo y cambian de dirección:
- Monedas: Si una moneda es recogida por n personaje, tanto el del usuario como el del ordenador, dicha moneda desaparece. Además, si un personaje es tocado por un enemigo, se reparten algunas de las monedas que tenía recogida por el nivel.
- Tiempo: El tiempo máximo para completar el nivel va corriendo hacia atrás, y es importante esta variable porque cuanto antes se llegue al final de la meta, más puntos se consiguen.

Por este hecho, es tan importante ir recalculando los movimientos a realizar mediante el algoritmo A*; para ir adaptándose en tiempo real a los estados del nivel. Se decía anteriormente que, la búsqueda de caminos se iba lanzando cada 3 ciclos de actualización (Update) o ticks. En el siguiente punto se va a desarrollar en detalle la explicación de este mecanismo:

4.3.6.3 Balanceo algoritmo

La idea es que un algoritmo de búsqueda A* completo dure 3 ciclos de Update. El motivo de la optimización que se basa en la premisa de que las personas no son capaces de ver 30 veces por segundo, como los frames por segundo del juego. Este hecho permite dedicar más ciclos para el A* y así poder obtener mejores resultados. En este caso se ha escogido que el algoritmo A* se ejecute durante 3 ciclos. Por tanto, el algoritmo se ejecuta durante 15 milisegundos cada ciclo. Notar que cada ciclo son 1/30 segundos, es decir, unos 33 milisegundos. Este valor se fija en la inicialización de la clase del juego, con la sentencia:

```
TargetElapsedTime = TimeSpan.FromTicks(333333);
```

El algoritmo A*, a parte de sus ventajas como que es un algoritmo de búsqueda completo, es decir, que si existe solución la encuentra siempre, se caracteriza también porque es muy costoso en recursos. Ocupa muchísima memoria, ya que es un algoritmo que va generando muchos hijos en su árbol de solución; y hay que ir procesando y guardando todos en sus listas. Por ello, se convirtió en objetivo de este proyecto la optimización de este proceso. Otro objetivo paralelo que se persigue también es su comportamiento realista.

La motivación principal para desarrollar esta mejora fue optimizar el tiempo que tiene el juego para realizar el Update de JugadorIA. Recordemos que XNA trabaja del siguiente modo: una vez que se ha inicializado y cargado el juego, XNA se encarga de gestionar la actualización (Update) y el dibujo por pantalla (Draw). Para cada uno de estos procesos, XNA se encarga de asignarle el tiempo disponible. Por ejemplo, hay muchas más actualizaciones en un segundo que dibujos por pantalla. Es decir, que si por ejemplo el juego corre a 30 fps (30 frames por segundo), aparte de dibujar en un segundo tantas veces, se realizan muchas más actualizaciones de la lógica del juego. Si cualquiera de las partes se retrasa, da lugar a un efecto cadena en la que todo se va retrasando. Como se ha comentado anteriormente, el algoritmo A* es muy pesado, y si esta carga no se balancea correctamente en el Update, la consecuencia es que todo el juego se retrasa y corre muchísimo más despacio.

Para balancear el algoritmo en 3 ciclos, se utiliza la variable contador de la clase JugadorIA. Dicha variable se encarga de ir llevando el control de las llamadas a EntradaAccionBot. Cada vez que es llamado, suma una unidad a su valor y hace el módulo entre 3:

```
contador = (contador + 1) % 3;
```

Con esto se consigue que, cada 3 llamadas, tengamos el valor 0 en *contador*. Y este valor es fundamental para el método *GenerarCamino* de la clase BusquedaAEstrella. Este es el método encargado de lanzar el algoritmo, dentro de su clase. Se le pasa como parámetro el nivel y el tiempo, así como una variable por referencia indicando si se ha encontrado

el final y la variable contador (aquí llamada pTick). El valor de pTick determina el estado del algoritmo:

- Si pTick es 0, se genera un nuevo algoritmo A*.
- Si pTick es distinto de 0 (1 ó 2), se continúa con el algoritmo que está a medias.

En la siguiente figura se diagrama el proceso:

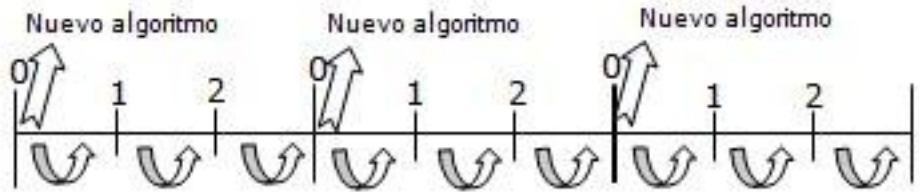


Ilustración 77 - Balanceo algoritmo

Cada vez que se genera un nuevo algoritmo, se reconstruye la lista de movimientos que se ha obtenido en el anterior algoritmo, el que acaba de concluir. El método *GenerarCamino* lo que devuelve es la lista de movimientos obtenidos como conclusión del algoritmo. Por tanto, si estamos en un ciclo distinto que 0, devolverá la lista de movimientos obtenida anteriormente; y, no es hasta que se llegue de nuevo a tick igual a 0, cuando se reconstruye la nueva lista. Este hecho también está basado en la idea que intenta simular un comportamiento más realista, para que el ordenador no sea extremadamente rápido a la hora de cambiar de acción. Un jugador humano no podría hacerlo a tantísima velocidad (la que marque el ciclo entre Update) y es lo que se intenta simular cambiando de acción cada 3 ciclos.

Todo este proceso que se acaba de describir es así durante toda la partida del jugador IA. Sin embargo, la única excepción que se produce es en la primera ejecución. Al entrar por primera vez, no hay ninguna lista de movimientos ya hecha anteriormente la cual devolver. Por tanto, en este caso y al ser pTick 0, se inicializa un nuevo algoritmo A* con normalidad

pero *no* se genera una nueva lista de movimientos. Por tanto, el jugador IA en esta primera ejecución no haría ninguna acción. Este hecho no es nada relevante en el transcurso del juego dado la cantidad elevadísima de updates que se producen y el jugador humano ni se daría cuenta.

Indicar, también, que todo este mecanismo se puede llevar a cabo gracias a que el algoritmo A* ha sido modificado para guardar sus listas abiertas y cerradas en una variable propia de la clase BusquedaAEstrella. Es decir, cuando se finaliza un subciclo del algoritmo, no se puede perder todo el proceso que se ha hecho, tanto su lista abierta como su lista cerrada. Por ello, ambas variables se guardan; y, al llegar el siguiente ciclo, estas variables son cargadas. En el caso de que se lance una nueva búsqueda, ambas listas son inicializadas de nuevo.

El método que lleva toda la lógica del A* es *buscarAEstrellaOptimizado*. Es llamado desde, el explicado en el punto anterior, *GenerarCamino*. No se quiere ser reincidente con la explicación de cómo funciona el A*, porque ha sido ya comentado al principio de este apartado. Sin embargo, sí que se reflejan aquí determinados aspectos técnicos sobre su implementación:

Lo primero a comentar es que este método se comporta distinto si se va a trabajar un algoritmo A* nuevo, o si se va a continuar con uno ya trabajado. Afecta a la primera fase del método, en la que se inicializan las variables. Si es nuevo, lo cual se sabe por el parámetro por valor pNuevo), lo primero que se hace es *clonar* el nivel. Esta clonación de nivel se realiza para tener una copia en memoria del estado exacto del nivel. La copia se realiza para mantener una referencia del estado original, ya que en a lo largo del algoritmo se irán haciendo simulaciones del nivel y se ha de poder mantener el conocimiento de cuál es el verdadero estado, y no el simulado.

Para la clonación, se tiene un método en la clase Nivel en el cual se crea un nuevo nivel y se le va asignando a éste, los valores del nivel actual. Es importante destacar que con los objetos "complejos" (que no sean tipos

de variables primarios como por ejemplo enteros o strings), como la clase JugadorIA, hay que clonarlos también. Dicha clase, tiene otro método clonar que es a la que se llamada desde la clonación de nivel:

```
public Nivel Clonar()
{
        Nivel nivel = new Nivel(this);

    int anchoNivel = new int();
    anchoNivel = this._anchoNivelEnPixeles;
    . . .
  nivel._jugadorIA = this.JugadorIA.Clonar(this, this.JugadorIA.Posicion,true)
. . .

    return nivel;
}
```

Otra clonación que se realiza en el algoritmo es la correspondiente al nodo (clase NodoAEstrella). Desde ella, también se llama a la clonación el nivel, pues es una de las propiedades de las clases.

A partir de ahí, se entra en el bucle de recorrido del algoritmo A*. En este paso es donde se van generando los hijos, y gestionando las listas abiertas y cerradas.

Hay que controlar el tiempo que se ejecuta el algoritmo. Como se comentó anteriormente, el algoritmo A* devuelve el camino óptimo; pero, para ello, necesita ejecutarse completamente, es decir, explora todos los nodos que necesite hasta llegar a la meta y consume todo el tiempo que sea necesario. Sin embargo, este hecho de que tenga completo uso libre de recursos y tiempo no es viable en este proyecto, porque el ciclo Update (en el cual se ejecuta el A*) tiene un tiempo disponible prefijado. Por tanto, hay que controlar al algoritmo.

Para ello, se han implementado unos contadores de tiempo que tienen como objetivo ir midiendo los tiempos de ejecución del algoritmo. Se le asigna un tiempo máximo de ejecución por fase del algoritmo, la cual está determinada por la constante TIEMPO_EJECUCION_ALGORITMO. En este caso, tiene un valor de 15 microsegundos. Dentro del algoritmo, se tiene la variable *microsegundos* que va almacenando el valor total de la suma de

todos los ciclos para comparar si es menor que el límite prefijado, es decir, 15. Cuando es mayor, se da por finalizada dicha fase del algoritmo.

```
Double microsegundos = 0;
do
      {
         Stopwatch sw = new Stopwatch();
         sw.Start();

        . . .

        sw.Stop();
         long microseg = sw.ElapsedTicks / (Stopwatch.Frequency / (1000L *
1000L));//Convertir a microsegundos
               microsegundos += microseg;
       }
    while (((!encontrado) && (microsegundos <
TIEMPO_EJECUCION_ALGORITMO_A_EST * 1000)) && (pListaAbierta.Count > 0));
```

En el framento de código se puede observar cómo se utiliza la variable microsegundos para controlar el while. Dicha variable va sumando cada vez que se realiza un ciclo completo: el cronómetro (Stopwatch) se inicia al principio y se para al final, y se convierte la medición que haya hecho a microsegundos. El bucle también controla si se ha encontrado el final del camino (con la variable *encontrado)* y que la lista abierta no esté vacía, pues eso significaría que no hay solución al algoritmo y se generaría un bucle infinito.

Además, a lo largo del algoritmo, existen otros cronómetros para medir la eficacia del algoritmo. Estas mediciones de tiempo se encuentran en:

- Tiempo de clonación.
- Tiempo de generación de un vecino (hijo).
- Tiempo de generación de todos los vecinos (5).
- Tiempo de simulación.
- Tiempo inserción en lista.
- Tiempo por ciclo.

Posteriormente, en la sección de pruebas, se detallarán los resultados obtenidos y se hará un estudio sobre ellos.

4.3.6.4 Obtención nodos vecinos

Siguiendo con el algoritmo A*, el paso siguiente es generar todos los vecinos (también llamados hijos) del nodo actual, a través del método: O*btenerVecinos.* El objetivo de esta función es la generación de los nodos colindantes al nodo actual. Para obtener dichos nodos, lo que se hace es *simular* el nivel con una acción específica. Tales acciones son las siguientes:

- Desplazamiento hacia la derecha.
- Desplazamiento hacia la izquierda.
- Salto vertical sin desplazamiento.
- Salto vertical hacia la derecha.
- Salto vertical hacia la izquierda.

De este modo, se generan 5 nodos vecinos en función del movimiento del personaje. Al principio de este tema, se hablaba del A* en función de casillas, como ocurría por ejemplo en los juegos de estrategia, y cómo se modelizó para este proyecto en función de los bloques del nivel. Pero también se comentó, que el movimiento entre bloques no era brusco, sino progresivo en función del desplazamiento del personaje. Y es justo en este punto donde se ve exactamente cómo se realiza la transición en la creación nodos. No se pasa directamente de una casilla a otra, sino que se simula el movimiento del personaje, y a raíz de ahí, ya se puede clasificar en qué bloque se encuentra. La simulación del personaje se consigue gracias a la lógica del juego. Se le asigna una acción, y se lanza la lógica de actualización del juego (UpdateSimulación) para obtener el nuevo estado del juego. En el siguiente código se puede observar cómo se asignan las acciones y se simulan. Indicar que listaAcciones es una lista con todas las posibles acciones comentadas anteriormente, y que se van recorriendo todas a través de un bucle for indizado por la variable i.

```
nodoHijo.Nivel.JugadorIA.Accion = listaAcciones[i][0];
if (listaAcciones[i][1] == 1) //Salto
{
    nodoHijo.Nivel.JugadorIA.EstaSaltando = true;
}
```

Esta simulación es lanzada 3 veces. La razón por la que se simula esas 3 veces es porque se ha balanceado un algoritmo completo también en 3 fases. De este modo, durante todo ese tiempo, se estaría "ejecutando" la misma acción. El valor del número de iteraciones que se realiza (3) se halla en la constante NUMERO_ITERACIONES_SIMULADOR.

```
//Simulación
for (int j = 1; j <= NUMERO_ITERACIONES_SIMULACION; j++)
{
     nodoHijo.Nivel.UpdateSimulacion(pNodoActual.Nivel.InstanteGT,
tiempoSimulado, listaAcciones[i]);
}
```

Una vez que se ha realizado la simulación y se ha obtenido la posición simulada del jugador, se produce un redondeo de las coordenadas de dicha posición, utilizando la función *RedondearCoordenadas*. El motivo por el que se hace es para unificar estados del jugador. Si la posición entre dos estados es prácticamente la misma, es decir, diferenciado por 1 o 2 píxeles, de esta manera se consigue aunar el estado y que sea el mismo. Esto es importante para la eficacia del algoritmo, ya que si se tienen muchísimos nodos cuando prácticamente son iguales, A* funciona muchísimo peor porque la cantidad de nodos generados es demasiado grande para que el A* pueda encontrar una buena solución en poco tiempo. El redondeo de coordenadas se clasifica utilizando el siguiente criterio:

- Si el último dígito está entre 0 y 3, se redondea a 2.
- Si el último dígito está entre 4 y 6, se redondea a 5.
- Si el último dígito está entre 7 y 9, se redondea a 8.

A continuación vendría el proceso de asignación de coste y heurística, el cual será explicado en el siguiente subapartado.

El último paso dentro del proceso de generación de un vecino es insertarlo ordenadamente en la lista de vecinos, mediante el método

insertarOrdenado. Dicha lista de vecinos es la que se devuelve como resultado de la función *obtenerVecinos,* una vez que se han generado todos los hijos*.*

Con la obtención de los nodos hijos, el último paso que queda para completar un ciclo es la inserción de dichos nodos en la lista abierta y en la lista cerrada. No pueden existir nodos repetidos; en tal caso, se rechazaría su inserción. Para comprobar si dos nodos son iguales se hace mediante la comparación de la posición del jugadorIA. Inserción en dicha lista es ordenada por el coste (F(n)), quedando los valores con menor coste al principio.

El método también va controlando la profundidad del árbol, mediante la variable *profundidadArbol.* Esta propiedad también está presente en la clase NodoAEstrella, para indicar qué profundidad corresponde a cada nodo. Cada vez que se genera un nuevo nodo en el algoritmo, se le añade es añade su valor, el cual es la suma de una unidad a la profundidad de su padre. La profundidad del árbol será el valor máximo de

Con estas últimas instrucciones, se completaría un ciclo del algoritmo A*. En ese momento, se haría la verificación anteriormente comentada del while para comprobar que no se ha encontrado el final del camino, que la lista abierta tiene elementos, y que el tiempo de microsegundos es menor que el marcado como límite máximo de ejecución del algoritmo. Si todas estas condiciones se cumplen, se volvería a repetir el ciclo desde el principio, donde se selecciona el primer elemento de la lista abierta para introducirlo en la lista cerrada y continuaría.

Cuando se sale del ciclo, se realiza la reconstrucción del camino generado. En la variable nodoActual tenemos el resultado del mejor nodo que hemos encontrado; y, a partir de él, recursivamente se reconstruye todo el árbol hasta llegar al nodo padre. Toda esta lista es la que marca el mejor camino encontrado.

4.3.6.5 Costes y heurística

Una vez que se ha simulado la posición del jugador, se entra en una de las partes más importantes y fundamentales del A*, el cual es la asignación de costes y heurísticas.

Recordemos que la función del algoritmo A* se caracteriza por:

$$F(n) = g(n) + h(n)$$

Siendo:

- F = Valor total.
- G(n) = Coste hasta llegar al punto actual.
- H(n) = Heurística hasta el final.

Para calcular el coste, es decir, la parte G, lo que se hace es sumar el coste del nodo padre más la diferencia entre la posición del nodo hijo y el nodo padre. Dicha distancia es la normalización de su vector distancia. La instrucción que realiza todo este calcula es la siguiente:

```
nodoHijo.Coste = pNodoActual.Coste + (nodoHijo.Nivel.JugadorIA.Posicion - pNodoActual.Nivel.JugadorIA.Posicion).Length();
```

La capacidad del personaje para evitar enemigos se define en este punto. Si, al simular la acción, resultase que el jugador colisiona con un enemigo, lo que se hace es aumentar muchísimo el valor del coste:

```
nodoHijo.Coste = pNodoActual.Coste + (nodoHijo.Nivel.JugadorIA.Posicion - pNodoActual.Nivel.JugadorIA.Posicion).Length();
//Si colisiona con enemigos aumentamos el coste
if (ComprobarColisionEnemigos(nodoHijo.Nivel))
{
   nodoHijo.Coste += 9999999;
}
```

Lo que se consigue con este procedimiento es que la suma de ese nodo alcance valores muy elevados. De esta forma, cuando se inserte dicho nodo en la lista abierta, se inserta en las últimas posiciones; ya que está ordenada con los valores más bajos al principio (y los que primero se ejecutan y salen a la lista abierta). Por tanto, el algoritmo lo que va consiguiendo es evitar estos movimientos en los que resulta perdedor y hace que tenga un coste tan elevado.

En la siguiente figura se muestra un esquema de actuación ante casos así, para una mayor comprensión:

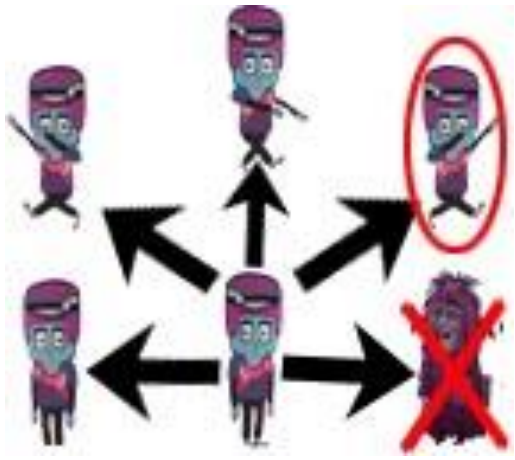


Ilustración 78 - Nodos vecino según movimiento personaje

La obtención de la heurística es uno de los factores claves para el éxito del algoritmo. El análisis y diseño de ésta ha tenido que hacerse desde cero, planteando el problema de la recolección de monedas, ya que no existía ningún precedente sobre el caso.

Destacar que en este tipo general de problemas, en el cual se quiere llegar de un punto a otro, se suele usar la distancia Euclídea como heurística, es decir, la distancia que falta para llegar al final. Sin embargo, en el caso de este videojuego queremos que el personaje recoja monedas para incrementar su puntuación final. Por ello, no se aplica la heurística de la distancia hasta la meta, sino esta distancia más una modificación en función de las monedas que el personaje pudieron conseguir siguiendo se hipotético camino.

Para calcula la heurística, se utiliza el siguiente procedimiento:

Primero se obtienen las monedas que están cerca del jugador IA. Se obtienen sólo las que están cerca porque si no la heurística consumiría demasiado tiempo. Para ello, se utiliza el método BuscarMonedas. Lo que hace este método es explorar los bloques alrededor del cual se encuentra el jugador, y comprobar si se encuentran monedas. Recorre 6 bloques verticalmente (3 en cada sentido) y 8 en dirección horizontal (4 en cada sentido). Va comprobando si en cada uno de esos bloques existe una moneda. Para ello, por cada bloque recorre todas las posibles monedas que hay en el mapa y comprueba si, efectivamente, colisiona:

```
if ((moneda.LimiteCirculo.Intersecciona(cuadro) && (moneda.Estado == 
EstadoMoneda.libre)))
        {
            insertarOrdenadoMoneda(moneda, ref listaPosicionesMonedas);
        }
```

Una vez se han obtenido la lista de monedas que están en el "espacio de acción" del personaje, se calcula la distancia desde la moneda más cercana al personaje. A partir de esta moneda, se van calculando las distancias de las demás monedas a la esta primera más cercana, y así sucesivamente, tal y como marca el siguiente algoritmo:

LISTA_MONEDAS = OBTENER MONEDAS CERCANAS
MODIFICAR HEURÍSTICA CON MONEDA MÁS CERCANA
DO
INTRODUCIR MONEDA EN LISTA DE YA VISITADAS Y SACARLA DE LISTA_MONEDAS
COGER SIGUIENTE MONEDA CERCANA
CALCULAR DISTANCIA DE ESA MONEDA AL PERSONAJE Y DISTANCIA A TODAS LAS DEMÁS MONEDAS
MODIFICAR HEURÍSTICA CON MONEDA VALOR DE DISTANCIA MENOR
WHILE (EXISTAN MONEDAS EN LISTA_MONEDAS)

Para calcular la modificación de la heurística, se utiliza el siguiente algoritmo; el cual está basado en la idea de relacionar la puntuación de una moneda con la unidad que utilizamos para la heurística y el coste, es decir, la distancia:

t1 = d / v (tiempo que se tardaría en coger la moneda)

t2 = p / q (tiempo que se gana por la puntuación al coger la moneda)

T = t1 - t2 (si este valor es negativo es que compensa coger la moneda)

Siendo:

- p = Puntuación de la moneda. Este valor es una constante por cada tipo de moneda.
- d = Distancia de la moneda al personaje
- v = Distancia recorrida en un segundo por el personaje en línea recta (en píxeles). Valor = 200;
- q = Puntos por cada segundo que llegues antes a la meta. Este valor es una constante. Valor = 50.

```
protected float CalcularHeuristica(Moneda pMon, float pHeuristica, Vector2 pPosicionJugador)
        {
            float t1 = ((pMon.Posicion - pPosicionJugador).Length()) / DISTANCIA_PIXELES_X_SEGUNDO;
            float t2 = pMon.Puntuacion / Nivel.PUNTUACION_SEGUNDO; //50

            float T = t1 - t2;
            float v = DISTANCIA_PIXELES_X_SEGUNDO;
            return pHeuristica + T * v;
        }
```

El propósito de este algoritmo es que si hay monedas cercas, el valor heurístico sea negativo. De esta forma, cuando se sume al coste para

obtener el valor total (f), la acción que aporte el obtener monedas quedará situada antes en la lista abierta; y, por tanto, se ejecutará antes.

Este algoritmo lo que hace es tener en cuenta también la distancia hasta el final de nivel y hasta la moneda; y cuanto tiempo tardaría en cogerlo. De esta manera, el algoritmo decide qué le conviene más, si coger la moneda o llegar al final. Además, también tiene en cuenta si hay muchas monedas alrededor, y por tanto le conviene ir hacia allí porque hay muchos puntos que recoger; o, sin embargo, si hay pocas monedas y de poco valor le conviene más ir hacia el final del nivel.

Las siguientes imágenes mostrarían un ejemplo de lo anterior:

En la primera imagen, al haber muchas monedas, el personaje volvería para atrás a recoger las monedas.

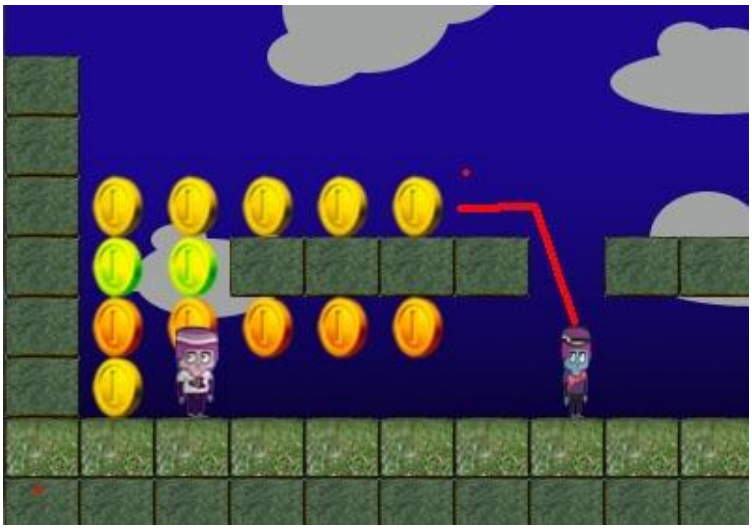


Ilustración 79 - Heurística recolección monedas 1

Sin embargo, en esta segunda imagen, al haber una moneda muy lejos hacia atrás, el personaje no decide volver a por ella y prosigue su marcha hasta el final del nivel, ya que le conviene más llegar pronto a la meta que coger esa moneda de tan baja puntuación.

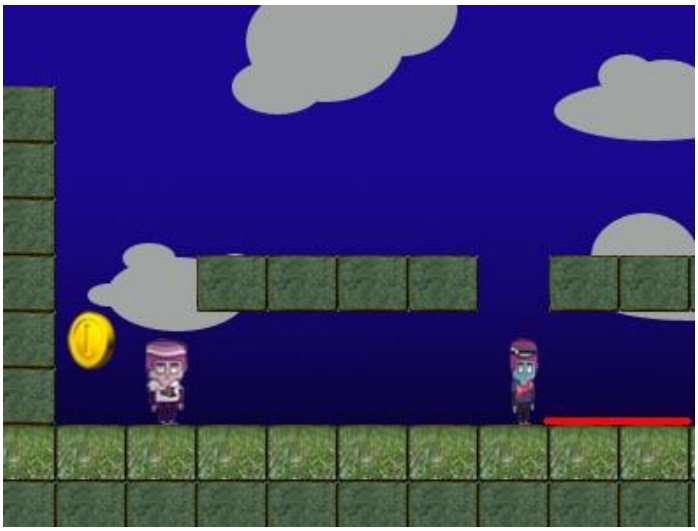


Ilustración 80 - Heurística recolección monedas 2

4.3.6.6. Optimizaciones

Como se ha comentado anteriormente, el algoritmo A* se caracteriza por ser muy pesado computacionalmente; y las tareas que se han llevado en cuanto a su optimización han sido fundamentales para su correcto funcionamiento:

- Destaca la optimización realizada en la lista abierta y la lista cerrada. En concreto, para la tarea de comprobación de si un nodo estaba ya insertado en dicha lista o no., se ha optado por usar una hash table, en lugar de recorrer toda la lista e ir comprobando uno a uno. Una hash table (o hash map) es una estructura de datos que usa unos valores clave para clasificar y ordenar los objetos. Actúa, por tanto, en cierto modo como un índice. Proporciona un acceso muchísimo más rápido que usando listas "normales". Para el proceso de indización se utiliza una función hash.

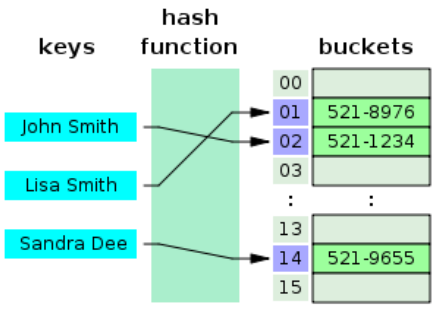


Ilustración 81 – Ejemplo hash table

Una hash table no puede tener dos claves iguales. Por tanto, con esta comprobación podemos saber si un nodo ha sido insertado ya o no:

```
if (!pHashTable.ContainsKey(CalcularHash(nodoActual)))
```

Si el índice existe en la table, significa que ese valor ha sido ya insertado y por tanto, no puede volver a introducirse. En caso contrario, se puede insertar con normalidad ya que no existe en la lista.

El método que calcula el hash para el nodo es CalcularHash. Está basado en la idea de aplicar un hash en función de la posición del jugador, y de la velocidad de su movimiento. En cuanto a su posición, se toma tanto la coordenada X y la coordenada Y, y se divide entre 2. Pero, esto no era suficiente, porque no reflejaba la inercia del jugador. Por ejemplo, no es lo mismo que el personaje esté en la posición X,Y si lleva una trayectoria hacia la izquierda, o hacia arriba. Es en esta idea en la que se basa para añadir un dígito al hash que indique la dirección del movimiento. Para ello, se usa un byte en el que se irán modificando sus bits en función de la velocidad. Al ser un byte, hay que convertirlo a entero para sumárselo al hash. A continuación se muestra el código para un mayor entendimiento:

```
private float CalcularHash(NodoAEstrella pNodo)
        {
            float hash = ((pNodo.Nivel.JugadorIA.Posicion.X / 2) *
pNodo.Nivel.AnchoNivelEnPixeles + (pNodo.Nivel.JugadorIA.Posicion.Y / 2));

            //Le añado la inercia;
            BitArray bitArr = new BitArray(8, false);

            Vector2 vel = pNodo.Nivel.JugadorIA.Velocidad;
            //[0] = derecha
            if (vel.X > 0)
                bitArr[0] = true;
            //[1] = izquierda
            if (vel.X < 0)
                bitArr[1] = true;
            //[2] = arriba
            if (vel.Y > 0)
                bitArr[2] = true;
            //[3] = abajo
            if (vel.Y < 0)
                bitArr[3] = true;

            int[] array = new int[1];
            bitArr.CopyTo(array, 0);
            int num = array[0];

            hash += num;

            return hash;
        }
```

Además, gracias a esta hashtable, se tiene registrado siempre que un nuevo nodo entre a la lista abierta. Si no se tuviese una hash table, cada vez que hubiese una nueva inserción, habría que comprobar tanto en la lista abierta como en la lista cerrada. Sin embargo, usando la hashtable no hace falta hacer esta comprobación lista a lista, sino que simplemente hay que comprobar que el hash no existe en la tabla.

- Otra optimización importante es la ordenación de la lista abierta. Se ha utilizado la técnica de ordenación por mezcla. El motivo de utilizar esta técnica es que está comprobado que mejora muchísimo los resultados de su rapidez; y es, por tanto, muchísimo más eficiente que insertar uno a uno los vecinos generados en la lista abierta. El procedimiento a ejecutar es la comparación entre dos listas previamente ya ordenadas. Va comprobando uno a uno entre las dos

listas, e introduce el de menor valor en otra lista auxiliar. Esta última lista auxiliar será el resultado de la ordenación de ambas. Las dos listas ya ordenadas que se tienen son, la lista de vecinos, y la lista abierta.

En la siguiente figura se representa un ejemplo: Habiendo obtenido la lista de vecinos, y teniendo también la lista abierta, ambas ya ordenadas, se procede a la ordenación por mezcla. Se va comprobando el primer elemento de ambas listas, y el que resulte de menor valor, es el que se añade a la lista auxiliar. El proceso se repite hasta que ambas listas quedan vacías.

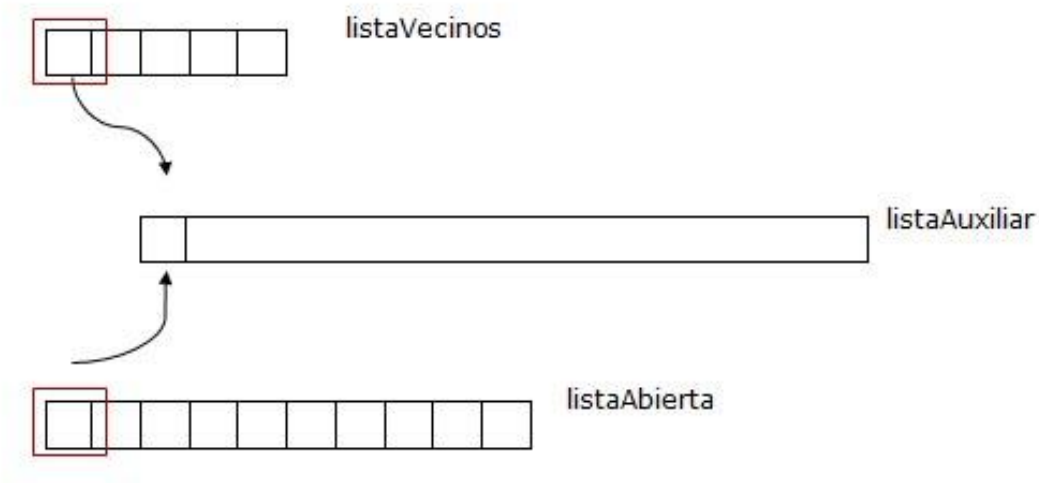


Ilustración 82 - Ordenamiento por mezcla

A continuación se muestra la codificación del algoritmo:

```
protected void ordenacionMezclada(ref List<NodoAEstrella> pListaAbierta,
List<NodoAEstrella> pListaHijos)
        {
            List<NodoAEstrella> nuevaLista = new List<NodoAEstrella>();

            int nuevaLongitud = pListaAbierta.Count + pListaHijos.Count;
            for (int a = 0; a < nuevaLongitud; a++)
            {
                NodoAEstrella nodo = new NodoAEstrella();
                nuevaLista.Add(nodo);
            }
            int i = 0;
            int j = 0;
            int n = 0;

            while ((n < nuevaLongitud) && (i < pListaAbierta.Count) && (j <
pListaHijos.Count))
```

```
            {
                while ((i < pListaAbierta.Count) && (pListaAbierta[i].CosteTotal <
pListaHijos[i].CosteTotal))
                {
                    nuevaLista[n] = pListaHijos[i];
                    i++;
                    n++;
                }
                while ((j < nuevaLongitud) && (pListaHijos[j].CosteTotal <
pListaAbierta[i].CosteTotal))
                {
                    nuevaLista[n] = pListaHijos[j];
                    j++;
                    n++;
                }
            }

            while (i < pListaAbierta.Count)
            {
                nuevaLista[n] = pListaAbierta[i];
                i++;
                n++;
            }
            while (j < nuevaLongitud)
            {
                nuevaLista[n] = pListaHijos[j];
                j++;
                n++;
            }

            pListaAbierta = nuevaLista;
        }
```

- Por último, se señala de nuevo el balanceo del algoritmo A* en 3 ciclos, técnica que ya ha sido explicado en el apartado anterior.

4.3.6.7. Trazas

La aplicación tiene la funcionalidad de guardar todas las trazas del algoritmo A* en un fichero de texto, para su posterior estudio de eficiencia y eficacia en función de los tiempos que emplea.

Todas las trazas se guardan en el directorio /Trazas, dentro de la carpeta en la que el juego haya sido instalado. El nombre del fichero se crea en función del día y la hora que se inicie la partida, siguiendo la siguiente forma:

DxHyyMzz.txt

Siendo las variables:

- x – Día de la semana
- yy – Hora de inicio de partida.
- zz – Minutos de inicio de partida.

El método que genera dichas estadísticas es *EstadisticasAlgoritmo*, de la clase BusquedaAEstrella. Se llama cada vez que una fase del algoritmo termina, antes de devolver la lista con los movimientos a realizar.

Se genera un fichero con la siguiente información:

- Fase del algoritmo - Inicial, intermedia o final. En función del tick de balanceo.
- Nodos generados en total, repetidos y porcentaje de colisiones (nodos repetidos) del algoritmo activo.
- Nodos generados en total, repetidos y porcentaje de colisiones (nodos repetidos) de ciclo actual.
- Número iteraciones del ciclo. – El número de veces que se ha repetido el bucle while del algoritmo.
- Tiempo total, máximo y medio empleado en la clonación de nodos.
- Tiempo total, máximo y medio empleado en la generación de nodos hijos.
- Tiempo total, máximo y medio empleado en la simulación de nodos hijos.
- Tiempo total, máximo y medio empleado en la inserción en lista abierta.
- Profundidad del árbol en el ciclo actual.
- Milisegundos totales empleados en el ciclo del algoritmo.

4.3.7 Sonidos

Los sonidos se reproducen usando al espacio de nombres del framework XNA Microsoft.Xna.Framewor.Audio. Toda la gestión de sonidos se lleva a cabo desde la clase Nivel. Hay que recordar que la reproducción de sonidos es una opción que puede elegir el usuario en el menú de Opciones; y que, por tanto, la ejecución o no de sonidos quedará en función de dicha elección. Este hecho se controla con la variable booleana ReproducirMusica.

Existen 2 tipos de sonidos fundamentales: la música de fondo y los efectos sonoros que se lanzan con determinados eventos.

En cuanto a la música de fondo se crea una instancia de sonido en el constructor del nivel. Dicha instancia sirve para gestionar el sonido, como por ejemplo su estado (reproduciéndose o parado) o el volumen. Además, en el constructor, se carga la música en su variable correspondiente (SoundEffect _musica):

```
_musica = Content.Load<SoundEffect>("Sonidos/Musica");
_musicaInstancia = _musica.CreateInstance();
```

A partir de ahí, es en cada ciclo de Update de la clase Nivel cuando se va controlando y ejecutando el sonido. Para ello, si, ReproducirMusica está activa, va verificando si su estado es parado, y si fuese así, volvería a reproducir. De esta manera, si el juego siguiese cuando hubiese acabado la canción, se produciría un bucle y volvería a empezar.

También, si el tiempo que falta es menor que el que indicar la constante WARNING_TIME, se acelera la velocidad de reproducción. El objetivo de esto es alertar al usuario que queda poco tiempo. Para ello, se cambia la propiedad Pitch de la instancia de música a 0.5 (normalmente es 0).

Todo este proceso se ejecuta gracias al siguiente código:

```
  //Música
  if (_reproducirMusica)
  {
        if (_musicaInstancia.State == SoundState.Stopped)
        {
             _musicaInstancia.Volume = _volumen;
             _musicaInstancia.Pitch = 0f;
             _musicaInstancia.Play();
        }
        if (TiempoRestante < GameplayScreen.WARNING_TIME)
        {
            _musicaInstancia.Pitch = 0.5f;
        }
   }
```

Por otro lado, los efectos sonoros se producen cuando el personaje es dañado por un enemigo, ha muerto o ha llegado al final de nivel. En cada uno de ellos, se reproducirá su sonido correspondiente con el método Play de la propiedad del sonido.

4.3.8 Opciones

La gestión de opciones se realiza desde la clase OptionsMenuScreen. El formato del fichero de opciones es XML, y su nombre es *AppConfig,* creándose en el mismo directorio que el ejecutable. La ruta relativa se obtiene mediante la función ObtenerRuta().

A partir de la carga del fichero, se procesan los valores de volumen y de música mediante la función cfgCargaValores.

```
private void CargarConfig()
{
       configXml.Load((ObtenerRuta() + "App.config"));
    volumen = int.Parse(cfgCargaValores("configuration/appSettings", "volumen"));
    if ((cfgCargaValores("configuration/appSettings", "musica")) == "Si")
    {
          musica = true;
    }
    else
    {
           musica = false;
    }
}

private string cfgCargaValores(string seccion, string clav
{
    XmlNode n;
```

```
    n = configXml.SelectSingleNode(seccion + "/add[@key=\"" + clave + "\"]");
    if (n != null)
    {
          return n.Attributes["value"].InnerText;
    }
    return "";
}
```

Cuando los valores son modificados y se sale al menú principal guardando estos cambios, lo que se hace es llama a la función guardarOpcionesMenuEntrySelected. Este método se encarga de guardar en el fichero de opciones xml los valores tratados (mediante el método cfgSetValue) que se están mostrando por pantalla:

```
cfgSetValue("configuration/appSettings", "volumen", volumen.ToString());
cfgSetValue("configuration/appSettings", "musica", ValorStringMusica(musica));

configXml.Save((ObtenerRuta() + "App.config"));
```

4.3.9 Gestión de puntuaciones

El sistema de gestión de puntuaciones se lleva a cabo en la clase PuntuacionesScreen.cs y HighScoreData.cs. El formato del fichero que contiene las puntuaciones es XML, y se carga desde el directorio Puntuaciones. Se carga dicho fichero y se introduce en un objeto tipo HighScoreData mediante la función LoadHighScores. De este modo, se tiene en memoria un objeto con todos los registros que han puntuado en el nivel en cuestión.

Si el jugador ha hecho una puntuación que entra en el ránking, debe introducir su nombre y pulsar Enter para guardar la puntuación. Este hecho se controla en el método Update mediante las propiedades booleanas salirMenuPrincipal y entraEnElRánking. Si entra en el ránking, cada vez que pulse una letra se va concatenando dicho carácter en la variable string nombre. Cuando el usuario pulsa Enter, se llama al método SaveHighScore, que lo que hace es guardar en el fichero de puntuaciones el nuevo ránking.

4.4. Pruebas

En este apartado se detallan las pruebas realizadas para verificar el correcto funcionamiento de la aplicación y que se cumplen los objetivos fijados.

4.4.1. Estudio trazas A*

En esta sección se expondrá algunas secciones de ejemplo de fichero de trazas resultante de la ejecución de una partida. Se pone sólo el resultado del algoritmo completo, aunque en el fichero de trazas se puede ver también clasificado por fases.

Se ha de destacar, también, ciertos aspectos que se han obtenido a lo largo del estudio de este proyecto. Los resultados de las trazas han sido obtenidas con el equipo de desarrollo (AMD x4 3.00 GHz; 4 GB RAM):

- El módulo de Inteligencia Artificial fue desarrollado en por separado, para tener una versión más simplificada, en la que se tenían únicamente los elementos imprescindibles. Por ejemplo, no se cargaban todas las texturas, sino que sólo las estrictamente necesarias. Los resultados que reflejaron las pruebas del módulo fueron muy buenos, y por ello se pasó a su implantación en el sistema global. Sin embargo, nos encontramos que al realizar nuevamente estas mediciones de pruebas arrojan resultados peores. La razón es que en esta versión completa, el juego tiene mucho más gasto computacional: como por ejemplo la gestión entre pantallas, mucho más contenido multimedia cargado, etc.

- El algoritmo gasta bastante tiempo también en la generación del fichero de trazas. Cada vez que se realiza una fase del algoritmo, se produce un acceso, escritura y guardado al fichero. Hay que tener en cuenta que, si este producto saliese en una versión final al mercado, no se incluiría este modo de "debug"

en el que genera siempre las estadísticas del algoritmo por medio de trazas y que, por tanto, el algoritmo obtendría mejores resultados.

- Por el motivo en el anterior punto señalado, se obtienen un número de nodos ligeramente bajo. Sin embargo, la aplicación y su IA funciona correctamente y por ello se ha dejado así.

- Todos estos resultados también dependen, en gran medida y como es lógico, del equipo sobre el que se ejecute. En el ordenador en el que se ha desarrollado, ha funcionado siempre muy bien y se le podían incluso meter más cómputo aumentando el tiempo de las iteraciones. Sin embargo, al pasar estos ejecutables a ordenadores más lentos, se obtenían resultados peores incluso llegando a ralentizar el juego. Por este motivo, se ha dejado en un tiempo de iteración de 15 milisegundos, porque funciona correctamente en todos los ordenadores probados.

Prueba 1:

~Nodos generados total algoritmo: 130
~Nodos repetidos algoritmo: 43
~Porcentaje colisiones total algoritmo: 33%
:Número iteraciones/ciclo: 35
!Tiempo total clonación de nodos: 0,221
!Tiempo medio clonación de nodos: 0,00491111111111111
·Tiempo total generación hijos: 16,177
·Tiempo medio generación hijos: 0,359488888888889
$Tiempo total simulación hijos: 0,554
$Tiempo medio simulación hijos: 0,0123111111111111
%Tiempo total inserción lista abierta: 0,114
&Tiempo total de ciclo: 16,369
&Tiempo medio de ciclo: 1,81877777777778
Profundidad arbol: 17
Milisegundos totales: 16,369

Prueba 2:

~Nodos generados total algoritmo: 120
~Nodos repetidos algoritmo: 27
~Porcentaje colisiones total algoritmo: 22%
:Número iteraciones/ciclo: 31
!Tiempo total clonación de nodos: 0,169
!Tiempo medio clonación de nodos: 0,004225
·Tiempo total generación hijos: 15,695
·Tiempo medio generación hijos: 0,392375
$Tiempo total simulación hijos: 0,491
$Tiempo medio simulación hijos: 0,012275
%Tiempo total inserción lista abierta: 0,133
&Tiempo total de ciclo: 15,909
&Tiempo medio de ciclo: 1,988625
Profundidad arbol: 20
Milisegundos totales: 15,909

4.4.2. Pruebas Inteligencia Artificial

A continuación se mostrarán las pruebas realizadas para comprobar el funcionamiento del agente de Inteligencia Artificial. Las pruebas han sido realizadas en diferentes niveles con diferentes características. El objetivo es comprobar algoritmo A* y el comportamiento en el personaje del ordenador ante diferentes escenarios. Indicar que en todas las pruebas realizadas, la meta se encontraba a la derecha del todo del nivel.

4.4.2.1 Prueba IA 1.

Objetivo: Comprobar que la IA es capaz de detectar monedas que están a su izquierda, (y por tanto más alejadas de la meta); y que el personaje es capaz de evitar enemigos.

Descripción: En un nivel en el que el jugador inicia su partida con monedas a su izquierda, vuelve a recogerlas. Además, al saltar la pared, se encuentra ante un enemigo.

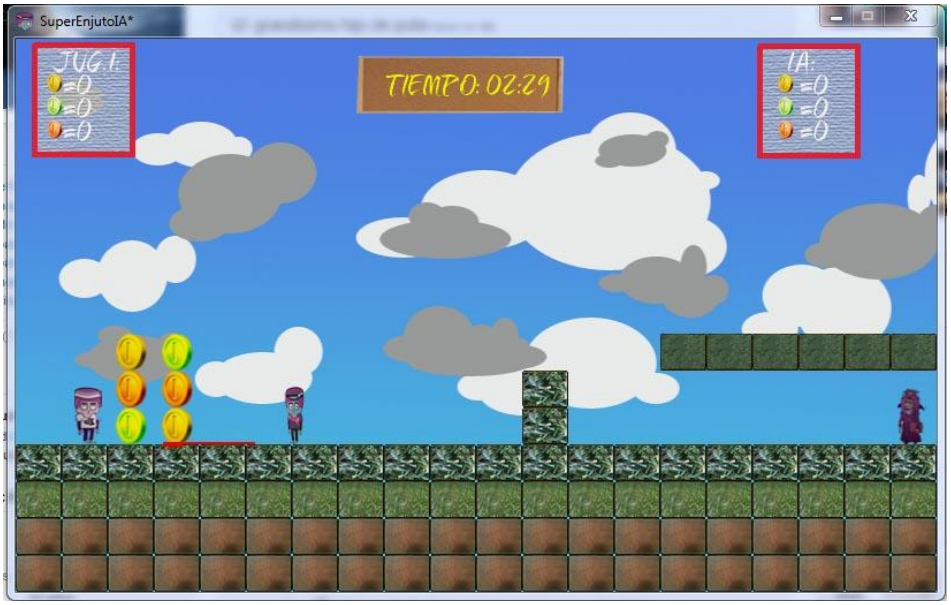


Ilustración 83 - Prueba 1.1

El personaje evita dicho enfrentamiento saltando sobre la plataforma superior y atravesándola.

Ilustración 84 - Prueba 1.2

4.4.2.2 Prueba IA 2

Objetivo: Comprobar que el personaje es capaz de subir a niveles superiores a por las monedas; y, que al bajar de dichos niveles, vuelve hacia atrás a por más monedas.

Descripción: El personaje puede avanzar en línea recta; o subir por la "escalera" y a la plataforma superior y continuar por ella. Esta última opción cuenta con monedas en su recorrido; así pues, es más conveniente y la IA elige este camino:

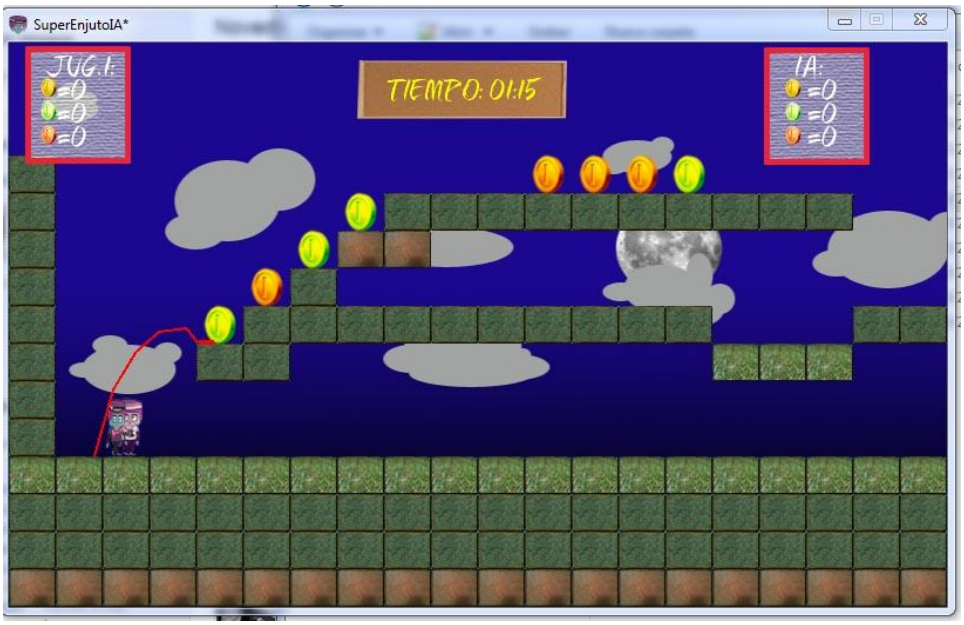


Ilustración 85 - Prueba 2.1

Al pasar la plataforma superior y llegar a la intermedia, detecta una moneda en el nivel inferior. En vez de seguir hacia adelante hasta el final, opta por bajar y volver hacia atrás a recoger la moneda.

Ilustración 86 - Prueba 2.2

Por último, en el suelo se encuentra ante un enemigo, por lo que decide saltarlo:

Ilustración 87 - Prueba 2.3

4.4.2.3 Prueba IA 3

Objetivo: Comprobar que el personaje es capaz de saltar por distintas plataformas en ambas direcciones para recoger las monedas.

Descripción: El personaje detecta monedas a su izquierda a un nivel superior, y salta a por ellas, en lugar de continuar recto hacia la meta.

Ilustración 88 - Prueba 3.1

A continuación, salta a la siguiente plataforma intermedia en busca de las siguientes monedas.

Ilustración 89 - Prueba 3.2

Detecta otra moneda en la plataforma superior, y salta hacia ella:

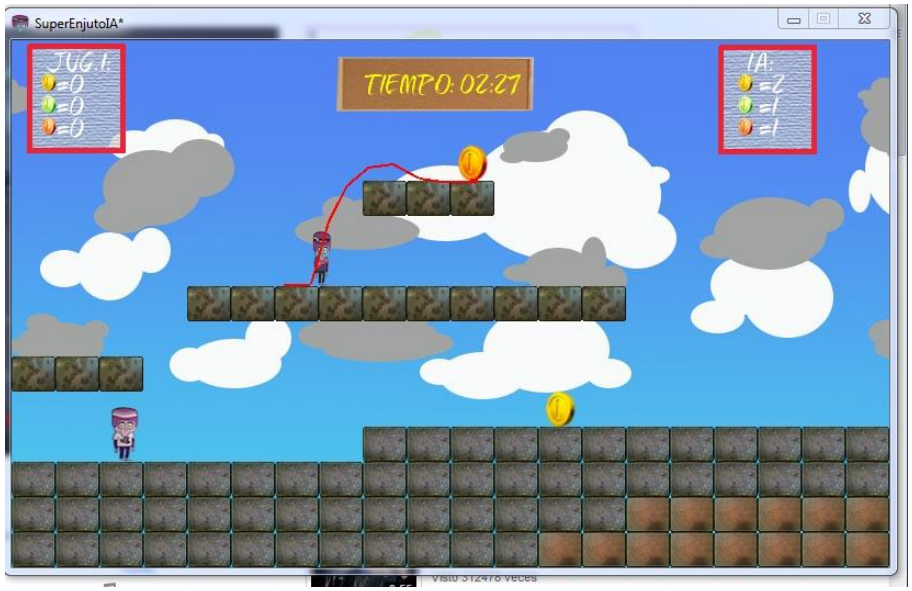


Ilustración 90 - Prueba 3.3

Por último, cuando va a bajar al suelo, detecta una moneda a su izquierda vuelve a recogerla.

Ilustración 91 - Prueba 3.4

4.4.2.4 Prueba IA 4

Objetivo: El personaje es capaz de salir de situaciones en las que tiene que echarse para atrás para poder avanzar, así como recoger

monedas en plataformas superiores que ya ha sobrepasado horizontalmente.

Descripción: El personaje evita los obstáculos y recoge monedas saltando primero hacia atrás para continuar dirección a la meta (derecha).

Ilustración 92 - Prueba 4.1

Detecta unas monedas a la izquierda en la "escalera" superior" izquierda y las recoge también:

Ilustración 93 - Prueba 4.2

Capítulo 5

Conclusiones

5. Conclusiones

Las conclusiones que saco, con el proyecto ya finalizado, son muy positivas. Se han cumplido, con creces, todos los objetivos prefijados al inicio del mismo. Lo que al principio parecía una tarea compleja, como era el hecho de crear una heurística que se comportase correctamente con un problema de tales características, se ha resuelto de una manera óptima y eficiente. Si bien es cierto que el resultado no es 100% perfecto, ya que si existen determinados escenarios en el que el personaje no actúa con absoluta inteligencia (por ejemplo cuando hay monedas muy separadas entre sí, o con niveles que requieren encadenar saltos precisos para alcanzar determinadas posiciones de monedas. No obstante, en la mayoría de los casos normales funciona correctamente.

He adquirido muchos conocimientos sobre el desarrollo de videojuegos; y, en concreto, de XNA. Con la primera fase, que consistió en leer dicha documentación sobre el tema, se adquirió una buena base. Estos conocimientos fueron bastante ampliados gracias a la gran comunidad de desarrolladores XNA Creators, la cual es una comunidad muy activa y participativa, donde se podían adquirir gran cantidad de recursos; así como obtener soporte y respuesta a las dudas que iban surgiendo. Además, el hecho de desarrollar en XNA implica un consecuente directo: programar en lenguaje C#. Al principio costó un poco, ya que es un lenguaje orientado a objetos y en la carrera (Ing.Inf.Técnica de Gestión) sólo hemos dado este paradigma teóricamente; realizando todas las prácticas de programación en lenguajes estructurados como Pascal o C. Sin embargo, este punto más que resultar un hándicap resultó una motivación añadida, ya que las asignaturas que más me han gustado a lo largo de la carrera han sido, sin lugar a dudas, las orientadas a la programación. Este proyecto me ha servido, por tanto, como un grandísimo aprendizaje sobre POO.

Otra de las conclusiones en claro que saco es el conocimiento adquirido sobre la Inteligencia Artificial aplicada a los videojuegos; y, en concreto, al algoritmo A*. Al cursar la asignatura *Inteligencia Artificial* en la carrera se ven diferentes técnicas, pero no se profundiza en detalle con ninguna. Gracias a este proyecto he podido profundizar en el algoritmo A*; y descubrir todo el potencial que tiene, así como los cuidados de optimización que hay que tener con él al aplicarlo a un problema real dado su alto coste en recursos.

Por otro lado, se han aplicado todos los conocimientos adquiridos sobre la Ingeniería del Software y la gestión de proyectos para planificar e ir gestionando éste.

Estoy también contento con el resultado final del producto. Es completamente jugable y aporta diversión al usuario. Además de la originalidad extra añadida al género de las plataformas, en concreto en su modo de juego compitiendo contra el ordenador.

Concluyo, por tanto, que estoy plenamente satisfecho con el resultado de este proyecto. Porque, aunque ha llevado muchísimo trabajo (y más de lo planificado en un principio al compaginarlo con el trabajo laboral y el estudio del Curso de Adaptación a Grado), considero que he adquirido unos conocimientos muy valiosos. Además, me gustaría orientar mi vida laboral hacia el mundo del desarrollo software; y, si pudiese ser, hacia el mundo de los videojuegos, ya que ha sido mi sueño desde niño. Este proyecto ha sido una gran base enriquecedora.

Capítulo 6

Trabajos futuros

6. Trabajos futuros

A lo largo del desarrollo de la aplicación, han ido surgiendo ideas que podrían servir para continuar añadiendo funcionalidades al proyecto. Se destacan las siguientes:

- ✓ *Generador de niveles "visual".* Aunque la aplicación permite crear nuevos niveles a partir de un fichero de texto plano, se podría añadir otro módulo en la aplicación con una interfaz gráfica destinada a tal efecto. Quedaría más bonito y visual para el usuario el poder ir viendo en tiempo real cómo quedaría realmente su nivel, en lugar de ir "imaginándoselo" al ir confeccionando el fichero de texto.

- ✓ *Generación automática de niveles.* Funcionalidad que crearía nuevos niveles al azar.

- ✓ *Xbox360.* Portar el juego a la plataforma Xbox 360. Tal y como se indicó en los sub-objetivos iniciales, el juego ha sido escrito dejando esta posibilidad abierta. Por ejemplo, en el proceso de registrar la entrada de usuario, aunque en el PC sólo se usa el teclado, el código tiene escrita su parte para aceptar el mando de la consola.

- ✓ *Añadir nuevos enemigos.* Además, estos enemigos tendrían nuevos movimientos, como por ejemplo ir saltando continuamente, o alternar aleatoriamente el sentido del movimiento.

- ✓ *Que el personaje pueda dañar a los enemigos.*

- ✓ *Modo historia.* Ir pasando de un nivel a otro y usar una historia con hilo argumental coherente con la temática del juego.

- ✓ *Modo cooperación.* Usar el algoritmo A* para, en vez de competir jugador humano y ordenador, trabajar juntos para llegar a la meta. Por ejemplo, que mientras que uno no haya accionado un pulsador, el otro no pueda avanzar; y viceversa.

Apéndices

A. Planificación

A.1. Planificación inicial

La duración planificada del proyecto fue de *116,25* días. Se planificó de tal manera para poder presentarlo en el mes de julio.

A continuación se detalla el diagrama de Gant planificado para el proyecto:

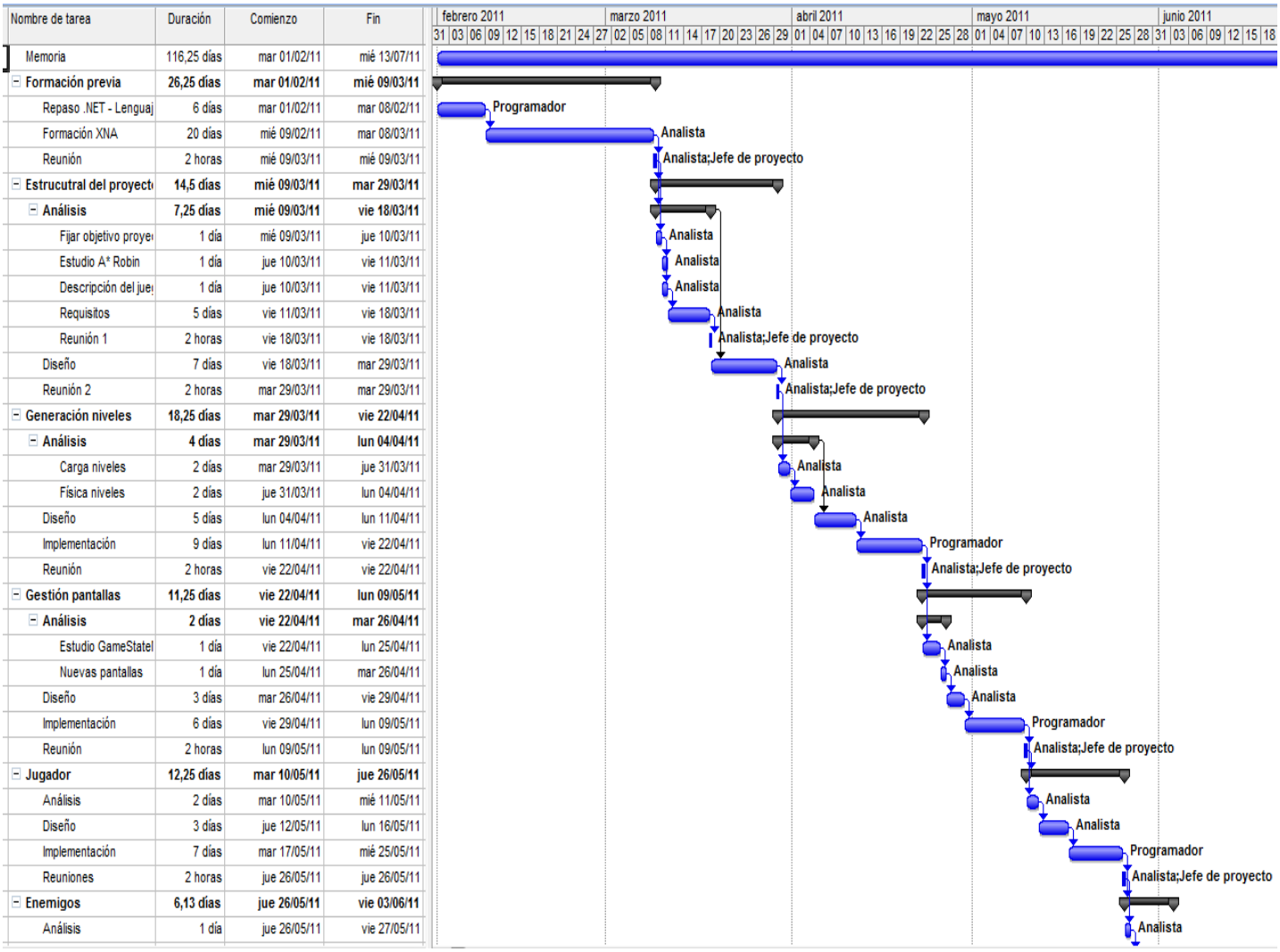
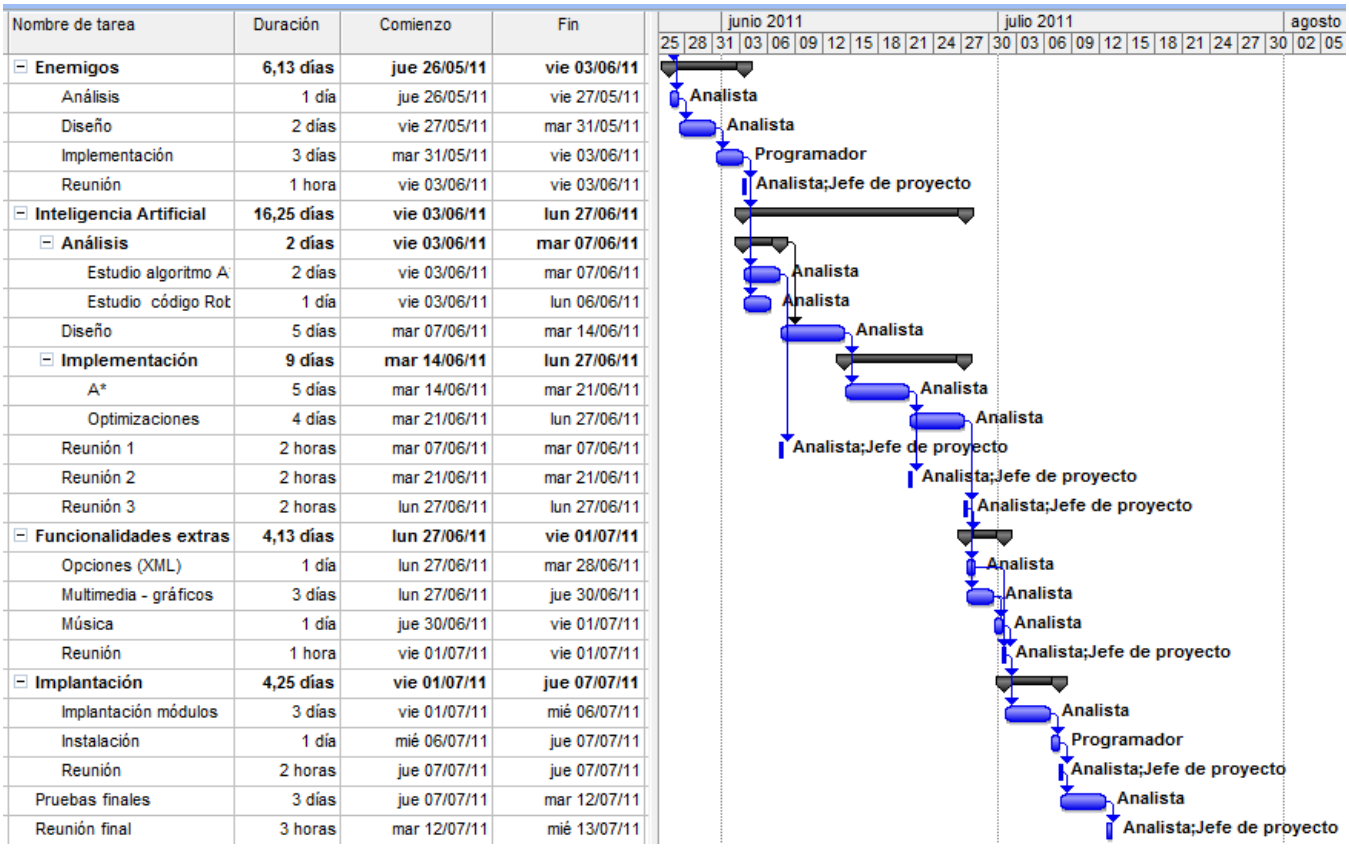


Ilustración 94 - Planificación uncial 1



Nombre de tarea	Duración	Comienzo	Fin
Enemigos	**6,13 días**	**jue 26/05/11**	**vie 03/06/11**
Análisis	1 día	jue 26/05/11	vie 27/05/11
Diseño	2 días	vie 27/05/11	mar 31/05/11
Implementación	3 días	mar 31/05/11	vie 03/06/11
Reunión	1 hora	vie 03/06/11	vie 03/06/11
Inteligencia Artificial	**16,25 días**	**vie 03/06/11**	**lun 27/06/11**
Análisis	**2 días**	**vie 03/06/11**	**mar 07/06/11**
Estudio algoritmo A	2 días	vie 03/06/11	mar 07/06/11
Estudio código Rob	1 día	vie 03/06/11	lun 06/06/11
Diseño	5 días	mar 07/06/11	mar 14/06/11
Implementación	**9 días**	**mar 14/06/11**	**lun 27/06/11**
A*	5 días	mar 14/06/11	mar 21/06/11
Optimizaciones	4 días	mar 21/06/11	lun 27/06/11
Reunión 1	2 horas	mar 07/06/11	mar 07/06/11
Reunión 2	2 horas	mar 21/06/11	mar 21/06/11
Reunión 3	2 horas	lun 27/06/11	lun 27/06/11
Funcionalidades extras	**4,13 días**	**lun 27/06/11**	**vie 01/07/11**
Opciones (XML)	1 día	lun 27/06/11	mar 28/06/11
Multimedia - gráficos	3 días	lun 27/06/11	jue 30/06/11
Música	1 día	jue 30/06/11	vie 01/07/11
Reunión	1 hora	vie 01/07/11	vie 01/07/11
Implantación	**4,25 días**	**vie 01/07/11**	**jue 07/07/11**
Implantación módulos	3 días	vie 01/07/11	mié 06/07/11
Instalación	1 día	mié 06/07/11	jue 07/07/11
Reunión	2 horas	jue 07/07/11	jue 07/07/11
Pruebas finales	3 días	jue 07/07/11	mar 12/07/11
Reunión final	3 horas	mar 12/07/11	mié 13/07/11

Ilustración 95 - Planificación inicial 2

A.2. Planificación final

La duración real del proyecto ha sido de *166* días.

El motivo del retraso fue la falta de tiempo al tener que compaginar con los estudios del Curso de Adaptación a Grado. Por este motivo, se fue acumulando un retraso el cual se recuperó en el mes de agosto, con las clases y exámenes ya finalizados, para poder presentar el proyecto en septiembre.

En las siguientes figuras se detalla el diagrama de Gant real del proyecto:

Nombre de tarea	Duración	Comienzo	Fin
Memoria	166 días	mar 01/02/11	mar 20/09/11
Formación previa	**32,25 días**	**mar 01/02/11**	**jue 17/03/11**
Repaso .NET - Lenguaje C#	7 días	mar 01/02/11	mié 09/02/11
Formación XNA	25 días	jue 10/02/11	mié 16/03/11
Reunión	2 horas	jue 17/03/11	jue 17/03/11
Estrucutral del proyecto	**17,5 días**	**jue 17/03/11**	**lun 11/04/11**
Análisis	**8,25 días**	**jue 17/03/11**	**mar 29/03/11**
Fijar objetivo proyecto	1 día	jue 17/03/11	vie 18/03/11
Estudio A* Robin	2 días	vie 18/03/11	mar 22/03/11
Descripción del juego	2 días	vie 18/03/11	mar 22/03/11
Requisitos	5 días	mar 22/03/11	mar 29/03/11
Reunión 1	2 horas	mar 29/03/11	mar 29/03/11
Diseño	9 días	mar 29/03/11	lun 11/04/11
Reunión 2	2 horas	lun 11/04/11	lun 11/04/11
Generación niveles	**25,25 días**	**lun 11/04/11**	**lun 16/05/11**
Análisis	**6 días**	**lun 11/04/11**	**mar 19/04/11**
Carga niveles	2 días	lun 11/04/11	mié 13/04/11
Física niveles	4 días	mié 13/04/11	mar 19/04/11
Diseño	6 días	mar 19/04/11	mié 27/04/11
Implementación	13 días	mié 27/04/11	lun 16/05/11
Reunión	2 horas	lun 16/05/11	lun 16/05/11
Gestión pantallas	**15,25 días**	**lun 16/05/11**	**lun 06/06/11**
Análisis	**3 días**	**lun 16/05/11**	**jue 19/05/11**
Estudio GameStateManag	1 día	lun 16/05/11	mar 17/05/11
Nuevas pantallas	2 días	mar 17/05/11	jue 19/05/11
Diseño	4 días	jue 19/05/11	mié 25/05/11
Implementación	8 días	mié 25/05/11	lun 06/06/11
Reunión	2 horas	lun 06/06/11	lun 06/06/11
Jugador	**16,31 días**	**mar 07/06/11**	**mié 29/06/11**
Análisis	3 días	mar 07/06/11	jue 09/06/11
Diseño	5 días	vie 10/06/11	jue 16/06/11
Implementación	8 días	vie 17/06/11	mar 28/06/11
Reunión	2,5 horas	mié 29/06/11	mié 29/06/11
Enemigos	**7,13 días**	**mié 29/06/11**	**vie 08/07/11**
Análisis	1 día	mié 29/06/11	jue 30/06/11

Ilustración 96 - Planificación final 1

Nombre de tarea	Duración	Comienzo	Fin
Enemigos	**7,13 días**	**mié 29/06/11**	**vie 08/07/11**
Análisis	1 día	mié 29/06/11	jue 30/06/11
Diseño	2 días	jue 30/06/11	lun 04/07/11
Implementación	4 días	lun 04/07/11	vie 08/07/11
Reunión	1 hora	vie 08/07/11	vie 08/07/11
Inteligencia Artificial	**21,25 días**	**vie 08/07/11**	**lun 08/08/11**
Análisis	**3 días**	**vie 08/07/11**	**mié 13/07/11**
Estudio algoritmo A*	3 días	vie 08/07/11	mié 13/07/11
Estudio código Robin A*	1,5 días	vie 08/07/11	lun 11/07/11
Diseño	5 días	mié 13/07/11	mié 20/07/11
Implementación	**13 días**	**mié 20/07/11**	**lun 08/08/11**
A*	5 días	mié 20/07/11	mié 27/07/11
Optimizaciones	8 días	mié 27/07/11	lun 08/08/11
Reunión 1	2 horas	mié 13/07/11	mié 13/07/11
Reunión 2	2 horas	mié 27/07/11	mié 27/07/11
Reunión 3	2 horas	lun 08/08/11	lun 08/08/11
Funcionalidades extras	**8 días?**	**lun 08/08/11**	**jue 18/08/11**
Opciones (XML)	2 días	lun 08/08/11	mié 10/08/11
Multimedia - gráficos	5 días	lun 08/08/11	lun 15/08/11
Música	2 días	lun 15/08/11	mié 17/08/11
Reunión	1 día?	mié 17/08/11	jue 18/08/11
Implantación	**5,25 días**	**jue 18/08/11**	**jue 25/08/11**
Implantación módulos	3 días	jue 18/08/11	mar 23/08/11
Instalación	2 días	mar 23/08/11	jue 25/08/11
Reunión	2 horas	jue 25/08/11	jue 25/08/11
Pruebas finales	5 días	jue 25/08/11	jue 01/09/11
Reunión final	3 horas	jue 01/09/11	vie 02/09/11

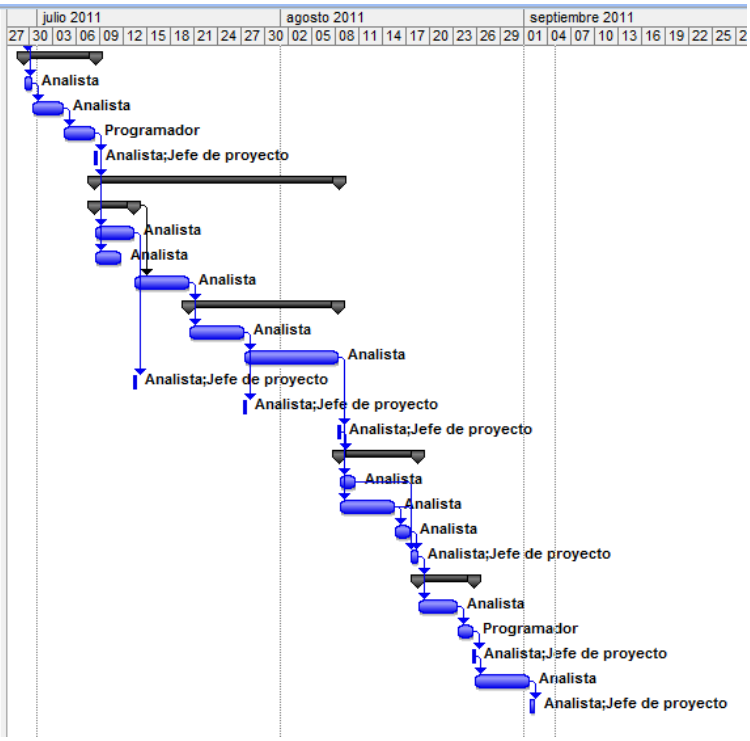


Ilustración 97 - Planificación final 2

B. Presupuesto

La tabla que contiene los recursos utilizados en el proyecto, así como su coste, es la siguiente:

Nombre	Puesto	Coste hombre mes
Jorge Fuentes Muñoz	Jefe de proyecto	3750 €
Jaime Chapinal Cervantes	Analista	2400 €
Jaime Chapinal Cervantes	Programador	1500 €

Tabla 69 - Presupuesto: coste recursos

B.1. Presupuesto inicial

Inicialmente, el proyecto estaba presupuestado según recogen las siguientes tablas:

- *Costes directos*

Costes de personal

Puesto	Dedicación	Coste total
Jefe de proyecto	0.19 hombres / mes	712
Analista	5.89 hombres / mes	14136
Programador	2.02 hombres / mes	3039
		17887

Coste de equipos

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable
Office Professional 2010	699,00	100	6	60	69,90
PC Windows 7	550,00	100	6	60	50,42
Portátil Asus	524,90	100	2	60	17,50

Adobe Photoshop CS 3	400,00	100	3	60	20,00
				Total	157.81

Otros costes directos del proyecto

Descripción	Empresa	Costes imputable
Viajes		40,00
Consumibles		15,00
Suministros eléctricos		70,00
Material oficina		20,00
	Total	145,00

Resumen de costes

A los costes indirectos se les aplica una tasa del 20%.

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	17.887
Amortización	158
Subcontratación de tareas	0
Costes de funcionamiento	145
Costes Indirectos	3.638
Total	21.828

El presupuesto inicial ascendía a la cantidad de *21.828* euros.

B.2. Coste real

Sin embargo, finalmente el coste ha sido según recogen las siguientes tablas:

- *Costes directos*

Costes de personal

Puesto	Dedicación	Coste total
Jefe de proyecto	0.31 hombres / mes	1162.5
Analista	7.89 hombres / mes	18936
Programador	2.56 hombres / mes	3840
		23938.5

Coste de equipos

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable
Office Professional 2010	699,00	100	7	60	81,55
PC Windows 7	550,00	100	7	60	64,17
Portátil Asus	524,90	100	2	60	17,50
Adobe Photoshop CS 3	400,00	100	3	60	20,00
				Total	183,21

Otros costes directos del proyecto

Descripción	Empresa	Costes imputable
Viajes		50,00
Consumibles		15,00
Suministros eléctricos		70,00
Material oficina		20,00
	Total	155,00

Resumen de costes

A los costes indirectos se les aplica una tasa del 20%.

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	24.682
Amortización	183
Subcontratación de tareas	0
Costes de funcionamiento	155
Costes Indirectos	5.004
Total	30.024

El coste total de este proyecto asciende a la cantidad de *30.024* euros.

B.3. Desviación

A continuación se muestran los datos de la desviación económica entre los datos presupuestados y los datos finales de coste.

Gráfica resumen:

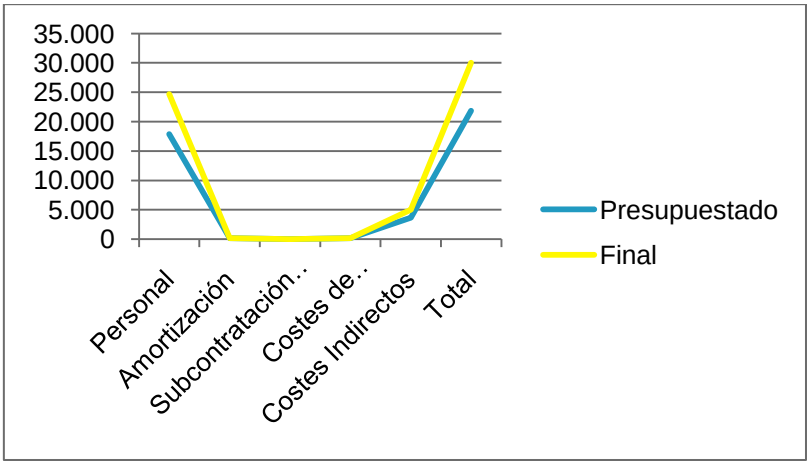


Ilustración 98 - Desviación económica

Se ha producido un *aumento* de costes del 37.54 %.

B.4. Amortización

El juego saldría a la venta por un precio de 5 €, al ser novedad. Se provee un volumen de ventas de 3.000 unidades en los 3 primeros meses. Se tendrían, por tanto, amortizados 15.000 €.

Durante los siguientes 4 meses, el precio bajaría a un valor de 3 €. Con tal precio, se estima unas ventas de 4.500 unidades (13500€) en los siguientes 5 meses. Hasta ese momento se tendrían 28500€ amortizados.

A partir de ese momento, el juego pasaría a costar 1 € ya que sería más antiguo. Quedarían por amortizar 1524 euros. Se estima de nuevo que tal cantidad de unidades estarían vendidas en 3 meses.

A modo de resumen, el juego tardaría 11 meses en ser totalmente amortizado.

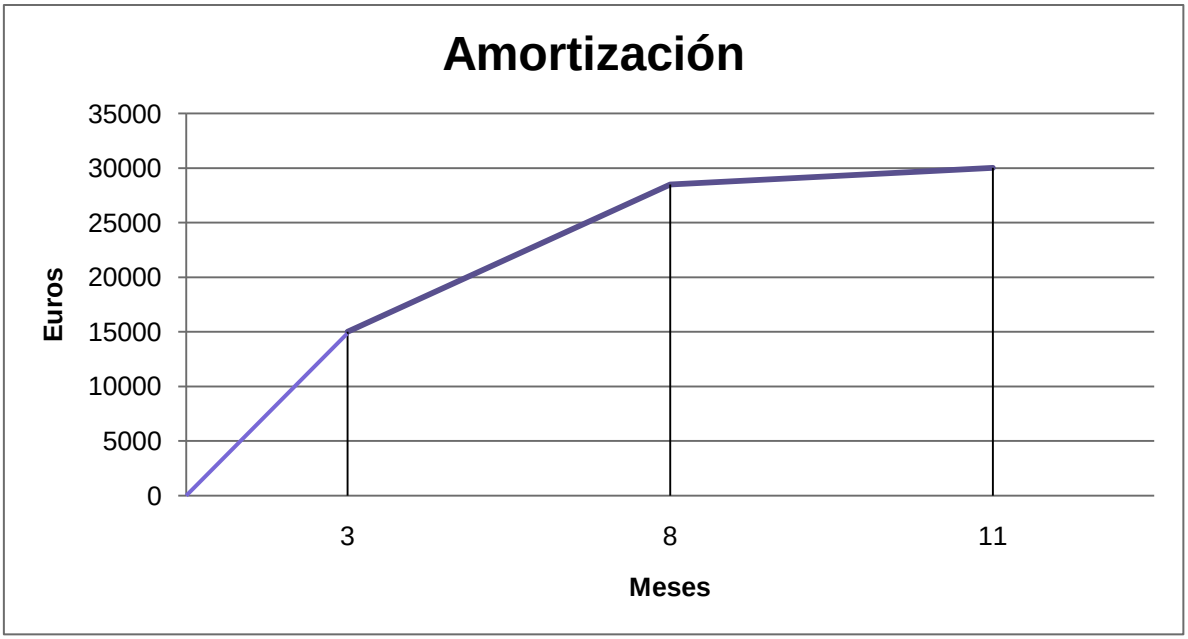


Ilustración 99 – Amortización

C. Manual de usuario

C.1. Requisitos previos.

El software mínimo que el ordenador ha de tener instalado es el siguiente:

- .Net Framework 4 – Framework del entorno Microsoft. Se puede descargar en la siguiente dirección: http://go.microsoft.com/fwlink/?linkid=182804
- XNA Framework 4.0 Redistributable – Framework del entorno XNA. Se puede descargar en la siguiente dirección: http://www.microsoft.com/download/en/confirmation.aspx?id=20914

C.2 Instalación.

Para instalar, se pulsa en el icono Setup.exe, apareciendo la siguiente pantalla:

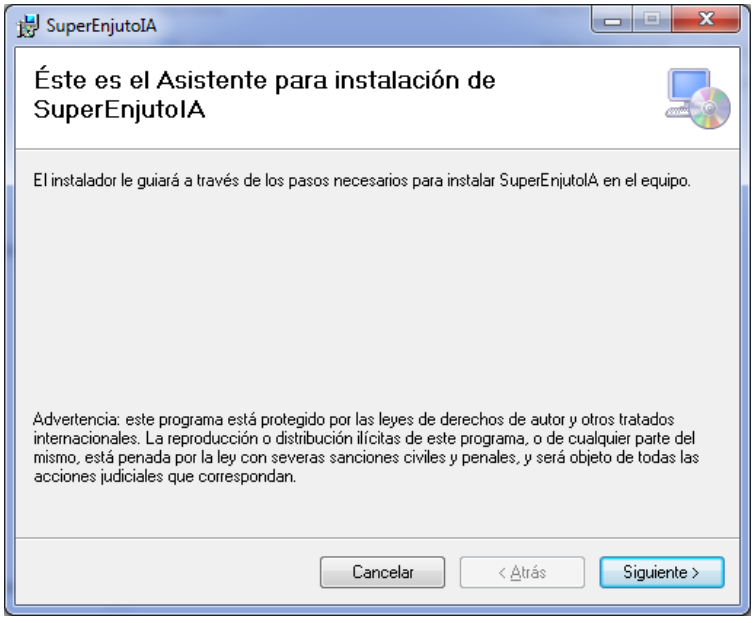


Ilustración 100 - Instalación 1

En la siguiente pantalla, se selecciona la ruta del disco duro donde el juego será instalado. Es importante elegir bien esta ruta porque será allí donde se guardarán los ficheros de niveles, así como los de puntuación y trazas del algoritmo A*.

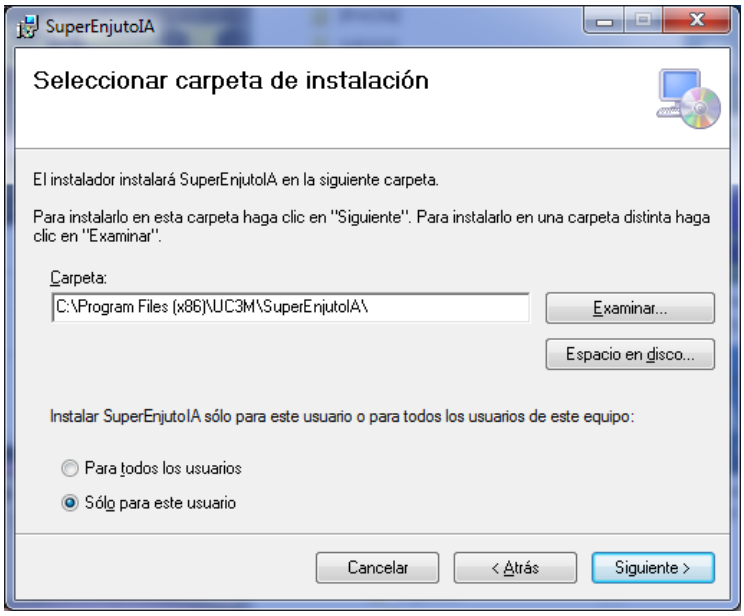


Ilustración 101 - Instalación 2

Una vez que el proceso de instalación ha finalizado, se crea un acceso directo tanto en el escritorio como en el menú inicio.

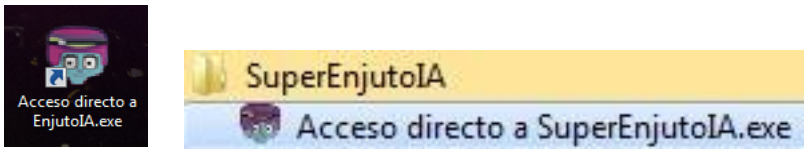


Ilustración 102 - Accesos directos aplicación

C.2 Manual de uso

Tras iniciar la aplicación, aparece el menú principal con las siguientes opciones:

- Jugar – Se inicia la partida del nivel 1.

- Seleccionar nivel – Se muestra una lista con los niveles disponibles para jugar.
- Mostrar puntuaciones – Se muestra un menú con una lista con los niveles disponibles para consultar la puntuación.
- Opciones – Se muestra el menú de opciones.
- Salir – Se sale de la aplicación.

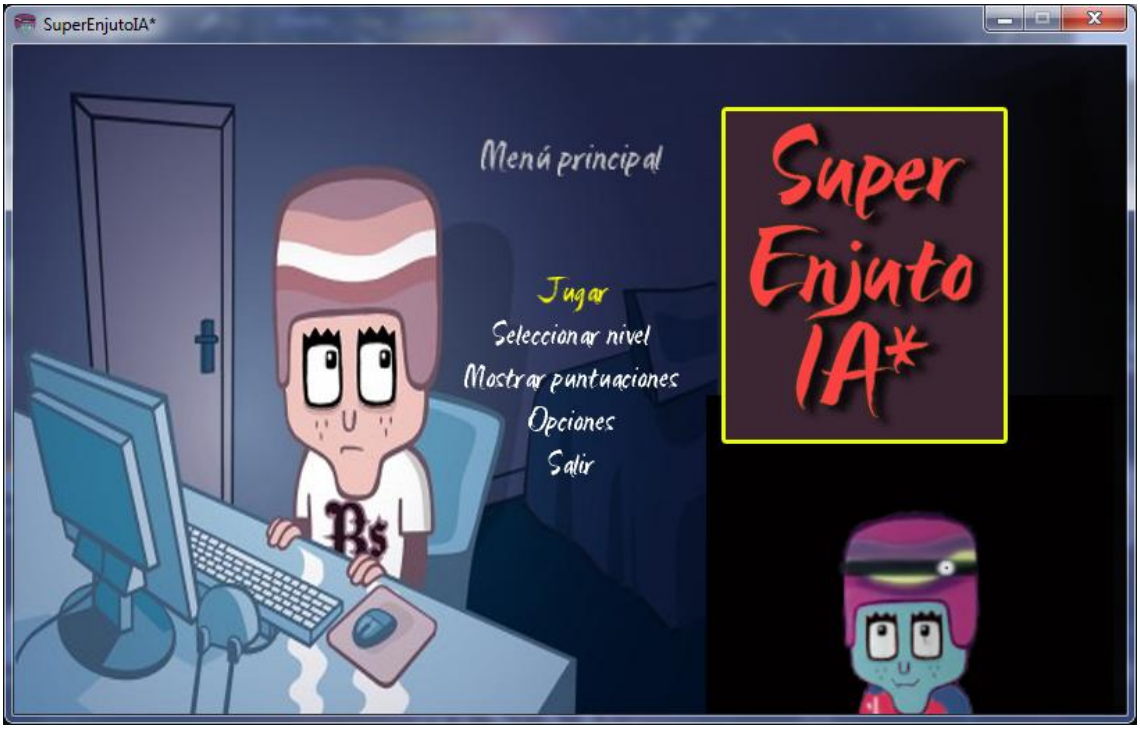


Ilustración 103 - Menú principal

JUGAR PARTIDA

Cuando se selecciona en el menú principal "Jugar"; O, AL SELECCIONAR UN NIVEL EN EL MENÚ Selección nivel, se carga una nueva partida de juego.

Los controles son los siguientes:

- *Desplazamiento izquierda* – Flecha izquierda
- *Desplazamiento derecha* – Flecha derecha.
- *Salto* – Barra espaciadora.

Al acabar una partida, se muestra la pantalla de fin de partida. En ella, se pueden observar las puntuaciones que ha hecho tanto el jugador humano o el jugador del ordenador; así como el que ha resultado ganador.

Ilustración 104 - Pantalla resultados

En esta pantalla, se muestra también un menú para salir directamente al menú principal o ir a la pantalla en la que se puede consultar las puntuaciones de otros jugadores en ese nivel. Si el jugador ha hecho una puntuación en la que entra al ránking, el usuario introduce su nombre y pulsa Enter para guardar.

Ilustración 105 - Pantalla mostrar puntuaciones

OPCIONES

En el menú opciones se pueden configurar la activación/desactivación de la música; como el volumen. Para cambiar las opciones, hay que pulsar Enter:

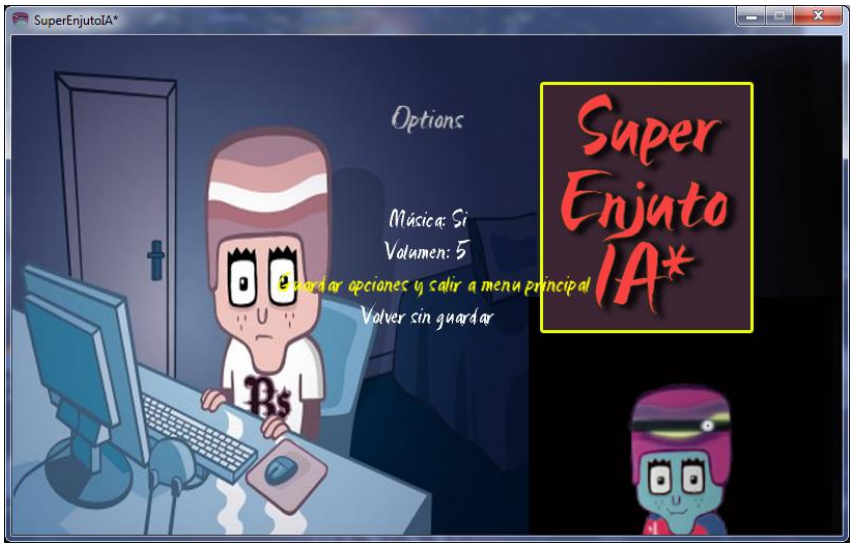


Ilustración 106 - Pantalla opciones

D. Medios empleados

El equipo con el que se ha desarrollado este proyecto es el siguiente:

- Procesador: AMD Athlon II x4 640, 300 MHz
- Memoria: 4 GB Memoria RAM (DDR3-1333 SDRAM)
- Tarjeta gráfica: ATI Radeon HD 4250
- Disco Duro: 500 GB Memoria RAM
- Sistema Operativo: Windows 7 Professional

En cuanto al software, se han utilizado los siguientes programas:

- Microsoft Visual Studio 2010 Professional
- XNA Framework 4.0
- DirectX 11.0
- Microsoft Office – Word, Excel, Power Point
- Microsoft Project

- Adobe Photoshop

Se ha utilizado XNA Framework 4.0, el cual tiene los siguientes requisitos mínimos:

- SS.OO: Windows XP, Windows Vista (SP 1) ó Windows 7.

- Tarjeta gráfica como mínimo Shader Model 1.1 y DirectX 9.0c. Se recomienda usar una tarjeta gráfica que permita Shader Model 2.0.

A parte del ordenador principal, se han utilizado dos portátiles complementarios. El primero de ellos se adquirió a mitad del proyecto, porque por cuestiones de tiempo no podía estar siempre en casa, donde estaba el ordenador principal. El uso que se le dio principalmente fue el desarrollo del código. Tiene las siguientes características:

- Asus A53S
- Procesador: i5 2.410Mhz
- Memoria: 4GB (DDR3 1333)
- Disco duro: 640 GB
- Tarjeta gráfica: 1GB
- Sistema Operativo: Windows 7 Home

Por último, se utilizó también un Netbook Acer Aspire. La finalidad de este equipo fue el manejo de documentos más fácilmente y con mayor portabilidad. También se usó para medir las optimizaciones y la eficacia de la aplicación con equipos de menores características, como es su caso:

- Procesador: Intel Atom N270 1.6 Ghz
- Memoria: 1GB
- Disco duro: 100 GB
- Sistema Operativo: Windows XP Home

Bibliografía y referencias

Bibliografía

- *Chad Carter,* "Microsoft XNA Game Studio 3.0 Unleashed", Ed. Sams. Año 2009.
- *Rob Miles,* "Introduction to Programming Through Game Development Using Microsoft XNA Game Studio", Academic Edition 2008
- *Stephen Cawood,* "Microsoft XNA Game Studio Express Creator's Guide", Ed: McGraw Hill , Año: 2007
- *Benjamin Nitschke,* "Professional XNA Game Programming: for Xbox 360 and Windows", Ed: Wiley . Año: 2007
- *Kurt Jaegers, XNA 4.0 Game Development by Example,* Ed: Packt Publishing, Año: 2010
- *Varios autores,* "AI Game Programming Wisdom", Ed: Steve Rabin. Año: 2002
- *Ian Millington; John Funge,* "Artificial Intelligence for Games – Second Edition";
- *Francisco Javier Ceballos Sierra,* "Microsoft C#: curso de programación"; Ed.: Ra-Ma; Año: 2006
- *Harvey M. Deitel,* "Cómo programar en C#", Ed. Pearson Educación; Año: 2007
- *Nils J Nilsson;* "Inteligencia artificial: una nueva síntesis"; Ed: McGraw-Hill; Año: 2000

Referencias web

[1] *MarioAI*, Web oficial, http://www.marioai.org/, [↑]

[2] *Robin B,* Vídeo Infinite Mario AI, http://www.youtube.com/watch?v=DlkMs4ZHHr8, [↑]

[3] *Robin B,* Código fuente Super Mario AI, http://www.doc.ic.ac.uk/~rb1006/projects:marioai [↑]

[4] *Alex J. Champandar,* Entrevista a Robin Baumgarten, http://aigamedev.com/open/interviews/mario-ai/ [↑]

[5] Web oficial Enjuto Mojamuto, http://www.enjutomojamuto.com/ [↑]

[6] *Alfonso,* Industria del Videojuego, un ejemplo de cómo salir adelante en crisis, http://himarketing.es/2011/08/industria-del-videojuego-un-ejemplo-de-como-salir-adelante-en-crisis/?utm_source=rss&utm_medium=rss&utm_campaign=industria-del-videojuego-un-ejemplo-de-como-salir-adelante-en-crisis, 1-Ago-2011 [↑]

[7] *Europa Press,*La industria del Videojuego factura un 10% más que en 2010, http://www.telecinco.es/informativos/tecnologia/noticia/250815/ 7-Jul-2011 [↑]

[8] *Martín Raposo, "*Programando por un sueño…", http://www.palermo.edu/economicas/pdf_economicas/pdfs_centrodeentretenimientosymedios/Articulo_videojuegos_v4.pdf, [↑]

[9] Web oficial ESRB, http://www.esrb.org/index-js.jsp [↑]

[10] *Webadictos,* Infografía Industria de los Videojuegos , http://www.webadictos.com.mx/2011/05/31/la-industria-de-los-videojuegos-infografia/ [↑]

[11] *Rosa Jiménez Cano,* "El videojuego español se reivindica como industria próspera", 14/0/2011 [↑]

[12] *Manuel Ángel Méndez,* "La industria española del videojuego busca cantera", http://www.cincodias.com/articulo/empresas/industria-espanola-videojuego-busca-cantera/20110715cdscdiemp_19/ 15/07/2011 [↥]

[13] *Daniel Escandell,* "EA: Wii U llegará en el momento perfecto para la industria", http://www.vandal.net/noticia/58444/ea-wii-u-llegara-en-el-momento-perfecto-para-la-industria/, 20/07/2011 [↥]

[14] *Wikipedia,* "Desarrollo de videojuegos independientes", http://es.wikipedia.org/wiki/Desarrollo_de_videojuegos_independiente , Últ.ed 20/02/2011 [↥]

[15] Web oficial Minecraft, http://www.minecraft.net/, [↥]

[16] Web oficial Plantas vs Zombies http://www.popcap.com/games/plants-vs-zombies/pc [↥]

[17] *Desconsolados,* Compra de Pop Cap, http://www.desconsolados.com/2011/07/12/electronic-arts-anuncia-compra-de-pop-cap-750-millones-de-dolares/ http://www.popcap.com/games/plants-vs-zombies/pc [↥]

[18] *Xbox369,* Indie Games, http://marketplace.xbox.com/es-ES/Games/XboxIndieGames [↥]

[19] *Nintendo,* WiiWare http://www.nintendo.es/NOE/es_ES/systems/wiiware_8343.html, [↥]

[20] *Apple*, iOS Dev Center, http://developer.apple.com/devcenter/ios/index.action , [↥]

[21] *Android,* Android Market, https://market.android.com/?hl=es, [↥]

[22] *Wiki ElOtrolado* http://www.elotrolado.net/wiki/G%C3%A9neros_de_los_videojuegos#Plataformas [↥]

[23] Wikipedia, "Plaform game", http://es.wikipedia.org/wiki/Platform_game [Último acceso Junio 2011] [↑]

[24] *Arcade-Museum*, "Space Panic" http://www.arcade-museum.com/game_detail.php?game_id=9676 [↑]

[25] *Wikipedia* "Juegos Arcade" http://es.wikipedia.org/wiki/Arcade [↑]

[26] *Nintendo*, http://www.nintendo.com/ [↑]

[27] *Arcade-History,* "Donkey Kong" http://www.arcade-history.com/?n=donkey-kong&page=detail&id=666 [Última edición 12 Mayo 2011] [↑]

[28] *The Dot Eaters*, "Enter the plumber", http://www.thedoteaters.com/p2_stage4.php [↑]

[29] *NinDB,* "Donkey Kong Jr.", http://www.nindb.net/game/donkey-kong-jr.html [↑]

[30] *FamicomWorld,* "Famicom World", http://famicomworld.com [↑]

[31] *NinDB*, "Popeye Famicom", http://www.nindb.net/game/popeye.html [↑]

[32] *Wikipedia, "NES",* http://en.wikipedia.org/wiki/Nintendo_Entertainment_System [↑]

[33] *Arcade Museum*, Jump Bug , http://www.arcade-museum.com/game_detail.php?letter=J&game_id=8250, [↑]

[34] *Wikipedia*, Side-scrolling, http://en.wikipedia.org/wiki/Side-scrolling_video_game, [↑]

[35] *joshnoodle,* "Pitfall", http://www.vinagreta.seriesmedia.org/basuco/?p=451 [1 Noviembre 2008], [↑]

[36] *Youtube,* Jungle Hunt, http://www.youtube.com/watch?index=1&feature=PlayList&v=oMwZ9zZNaPY&list=PL25C51137BB486AC6 [↑]

[37] *GameFaqs,* Smurf Rescue in Gargamel's Castle, http://www.gamefaqs.com/colecovision/585563-smurf-rescue-in-gargamels-castle, [↑]

[38]*GameSpectrum,* Jet Set Willy, http://www.worldofspectrum.org/infoseekid.cgi?id=0002589, [↑]

[39] *Wikipedia,* Moon Patrol, http://en.wikipedia.org/wiki/Moon_Patrol, [↑]

[40] *Wikipedia,* Parallax-Scrolling, http://en.wikipedia.org/wiki/Parallax_scrolling [↑]

[41] *Gamesradar,* "Gaming's most importante evolutions", http://www.gamesradar.com/gamings-most-important-evolutions/?page=3 [↑]

[42] *Wikipedia,* "Super Mario Bros.", http://en.wikipedia.org/wiki/Super_Mario_Bros. [↑]

[43] *Game Faqs,* "Alex Kidd", http://www.gamefaqs.com/sms/588027-alex-kidd-in-miracle-world [↑]

[44] *Hardcoregaming101,* "Wonder Boy", http://www.hardcoregaming101.net/wonderboy/wonderboy.htm, [↑]

[45] *Wikipedia,* "Metroid", http://en.wikipedia.org/wiki/Metroid_%28video_game%29, [↑]

[46] *Wikipedia,* “Megaman”, http://es.wikipedia.org/wiki/Mega_Man_%28NES%29 [↑]

[47] *Wikipedia,* “Super Nintendo”, http://en.wikipedia.org/wiki/Super_nintendo, [↑]

[48] *Wikipedia,* “Super Mario World”, http://www.mariowiki.com/Super_Mario_World, [↑]

[49] *Wikipedia,* “Sonic the Hedgehog”, http://en.wikipedia.org/wiki/Sonic_the_Hedgehog_(1991_video_game) , [↑]

[50] *Shikadi,* “Commander Keen”, http://www.shikadi.net/keenwiki/Main_Page, [↑]

[51] “Alpha waves”, http://hol.abime.net/3251 [↑]

[52] *Wikipedia,* “Jumping Flash”, http://en.wikipedia.org/wiki/Jumping_Flash!, [↑]

[53] “Bugs!”, http://www.pickhut.com/re/bugre.html, [↑]

[54] *Wikipedia,* “Crash Bandicoot”, http://es.wikipedia.org/wiki/Crash_Bandicoot_%28serie%29, [↑]

[55] *Sonic Retro,* “Sonic Adventure”, http://info.sonicretro.org/Sonic_Adventure, [↑]

[56] *A.L. Samuel*, “Some Studie in Machine Learning using the Game of Checkers”, https://researcher.ibm.com/researcher/files/us-beygel/samuel-checkers.pdf,

[57] *James Matthews,* “Arboles Minimax”, http://vidaartificial.com/index.php?title=Arboles_Minimax_%28Generation5.org%29, [↑]

[58] "Game AI:Strategy", http://www.sci.tamucc.edu/~sking/Courses/GameProgrammingGrad/Slides/GameAI.pdf, [↑]

[59] *Alex J. Champandard,* "Top 10 most influential AI Games", http://aigamedev.com/open/highlights/top-ai-games/, [↑]

[60] "Simspedia", http://es.sims.wikia.com/wiki/Los_Sims, [↑]

[61] *Alex J Champard,* "Living whit the Sims'Ai: 21 tricks to adopt for your game", http://aigamedev.com/open/highlights/the-sims-ai/, [↑]

[62] "Simulating and Modeling: Under the hood of The Sims", http://www.cs.northwestern.edu/~forbus/c95-gd/lectures/The_Sims_Under_the_Hood_files/v3_document.htm, [↑]

[63] *Jason Ocampo,* "Sims getting smarter", http://uk.pc.ign.com/articles/961/961065p1.html, [↑]

[64] "Robocup", http://www.robocup.org/, [↑]

[65] "Robocup Spain", http://www.robocupspain.es, [↑]

[66] "CAOS Coach Team", http://www.caos.inf.uc3m.es/caoscoachteam/index.htm, [↑]

[67] *B.López, M. Montaner, J.L. de la Rosa,* "Utilización de un simulador de Fútbol para Enseñar Inteligencia Artifical a Ingenieros", http://eia.udg.es/~mmontane/lopez-jenui01.pdf, [↑]

[68] "Robocup: IA en sistemas multiagente y fútbol de simulación", http://www.slideshare.net/jborregouses/robocup-inteligencia-artificial-en-sistemas-multiagente-y-ftbol-de-simulacin, [↑]

[69] *Eduardo Jiménez,* "PES 2012 vs FIFA 12: Comparativa entre sus sistemas de Inteligencia Artificial", http://www.gamerzona.com/2011/07/18/pes-2012-vs-fifa-12-comparativa-entre-sus-sistemas-de-inteligencia-artificial/, [↑]

[70] *Víctor Martínez, "*La IA de FIFA 12", http://hastalosjuegos.es/noticia/2187/la-ia-de-fifa-12/, [↑]

[71] "Videojuegos más difíciles y con mejores IA", http://www.3djuegos.com/foros/tema/723867/0/videojuegos-mas-dificiles-y-con-mejores-ia/, [↑]

[72] *Wikipedia,* "Cell", http://en.wikipedia.org/wiki/Cell_microprocessor, [↑]

[73] *Alexander Gambotto-Burke,* "The hard-thought race for intelligent gaming", [↑]

[74] *Wikia,* "Creatures", http://creatures.wikia.com/wiki/Creatures, [↑]

[75] *Larry Hardesty,* "Computer learns language by playing games", http://web.mit.edu/newsoffice/2011/language-from-games-0712.html, [↑]

[76] *Danilo Benzatti,* "Collective Intelligence", http://ai-depot.com/CollectiveIntelligence/Ant.html [↑]

[77] *Wikipedia,* "Microsoft XNA", http://en.wikipedia.org/wiki/Microsoft_XNA#XNA_Game_Studio_Express [Última edición 16 julio 2011] [↑]

[78] *Wikipedia,* "Framework", http://es.wikipedia.org/wiki/Framework [Última edición 2 agosto 2011] [↑]

[79] *Wikipedia, "CLR",* http://es.wikipedia.org/wiki/Common_Language_Runtime [Última edición 30 abril 2011] [↑]

[80] *Zune,* http://www.zune.net/es-es/ [↑]

[81] *Nielsen,* US Smartphone Market, http://blog.nielsen.com/nielsenwire/?p=28516 [28 Julio 2011] [↑]

[82] *Mauricio Jaramillo Marín, enter.co,* "Nokia y Microsoft" http://www.enter.co/movilidad/nokia-y-microsoft-juntos-para-luchar-contra-apple-y-google/ [11 Febrero 2001] [↑]

[83] *Wikipedia,* Symbian OS http://es.wikipedia.org/wiki/Symbian_OS [Última edición 28 Julio 2011] [↑]

[84] *Microsof Support,*What's New in XNA Game Studio 4.0, http://msdn.microsoft.com/en-us/library/bb417503.aspx [Última edición 5 Mayo 2011] [↑]

[85] *Nelxon Studio,Want to convert an XNA 3.1 project to XNA 4.0?* http://www.nelxon.com/blog/xna-3-1-to-xna-4-0-cheatsheet/ [↑]

[86] *Shawn Hargreaves,* Breaking changes in XNA Game Studio 4.0, http://blogs.msdn.com/b/shawnhar/archive/2010/03/16/breaking-changes-in-xna-game-studio-4-0.aspx [16 Marzo 2010] [↑]

[87] "MonoGame – Write Once, Play Everywhere", http://monogame.codeplex.com/, [↑]

[88] *Wikipedia,* ProyectoMono, http://es.wikipedia.org/wiki/Proyecto_Mono, [↑]

[89] *Wikipedia,* "OpenGL", http://es.wikipedia.org/wiki/OpenGL, [↑]

[90] "XNATouch", http://geeks.ms/blogs/jbosch/archive/2011/03/20/xna-conociendo-a-xna-touch.aspx, [↑]

[91] "Open Kinect", http://www.adafruit.com/blog/2010/11/04/the-open-kinect-project-the-ok-prize-get-1000-bounty-for-kinect-for-xbox-360-open-source-drivers/, [↑]

[92] "Winner open Kinect", http://www.adafruit.com/blog/2010/11/10/we-have-a-winner-open-kinect-drivers-released-winner-will-use-3k-for-more-hacking-plus-an-additional-2k-goes-to-the-eff/, [↑]

[93] *Javier Martín,* "Entrevista a Héctor Martín", http://www.elpais.com/articulo/tecnologia/gusta/trastear/elpeputec/20101111elpeputec_4/Tes, [↑]

[94] *Wikipedia,* Scene, http://es.wikipedia.org/wiki/Scene, [↑]

[95] "Web official Marcansoft", http://marcansoft.com/blog/, [↑]

[96] "Web oficial SDK Kinect", http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/default.aspx, [↑]

[97] "Documentación SDK Kinect", http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/docs/ProgrammingGuide_KinectSDK.pdf , [↑]

[98] *Josep Maria Sempere,* "Kinect tundra soporte XNA", http://www.eurogamer.es/articles/2010-11-24-kinect-tendra-soporte-xna , [↑]

[99] *Jesús Bosch,* "Experimentando Kinect + XNA (SDK official), http://geeks.ms/blogs/jbosch/archive/2011/06/16/experimentando-kinect-xna-sdk-oficial.aspx, [↑]

[100] *Carlos Chavez,* "Realizando pruebas con Kinect + XNA", http://geeks.ms/blogs/cchavez/archive/2011/04/20/realizando-pruebas-con-kinect-xna.aspx, [↑]

[101] "Kinect 3D head tracking with XNA 4.0 and OpenNI", http://www.kinecthacks.nl/2011/03/16/kinect-3d-head-tracking-with-xna-4-0-and-openni/, [↑]

[102] "Web official Dream.Build.Play", http://www.dreambuildplay.com/main/default.aspx, [↑]

[103] "Web official Rotor Scope", http://www.rotorscope-game.com/, [↑]

[104] – "Adventure Game Studio", http://www.adventuregamestudio.co.uk/, [↑]

[105] Allego, http://alleg.sourceforge.net/, [↑]

[106] Build Engine, http://advsys.net/ken/build.htm, [↑]

[107] 3D Game Studio, http://www.3dgamestudio.com, [↑]

[108] GameMaker, http://www.yoyogames.com/, [↑]

[109] Havok, http://www.havok.com/, [↑]

[110] "LWJGL", http://lwjgl.org/, [↑]

[111] "M.U.G.E.N.", http://www.elecbyte.com/, [↑]

[112] "Pygame", http://www.pygame.org/, [↑]

[113] "RPG Maker", http://www.rpgmakerweb.com/ , [↑]

[114] "AGI", http://www.agidev.com/, [↑]

[115] "Visionaire", http://www.visionaire-studio.net/, [↑]

[116] "Wintermute", http://dead-code.org/home/ , [↑]

[117] *XNA Creatos,* Game State Management, http://create.msdn.com/en-US/education/catalog/sample/game_state_management, [↑]

[118] "Música libre para juegos", http://entidad3d.comule.com/Musica/musica.htm, [↑]

[119] "XNA Platformer", http://create.msdn.com/en-US/education/catalog/sample/platformer, [↑]